MIXED-MEDIA FILE SYSTEMS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. F.A. van Vught,
volgens besluit van het College voor Promoties,
in het openbaar te verdedigen
op vrijdag 25 juni 1999 te 13.15 uur

door
Hendrikus Gerardus Petrus Bosch
geboren op 21 september 1966
te Amsterdam

Dit proefschift is goedgekeurd door de promotor prof. dr. S.J. Mullender.

Contents

Abstract **i**

Samenvatting **iii**

1 Introduction **1**
1.1 Mixed-media file systems . . . 3
1.2 Scope of this thesis . . . 8
1.3 Rationale . . . 8
1.4 Road map of this thesis . . . 9

2 Related work **11**
2.1 Existing UNIX file systems . . . 11
2.1.1 UNIX *Least Recently Used* (LRU) caches . . . 12
2.1.2 UNIX inodes . . . 12
2.1.3 UNIX file-system layouts . . . 13
2.2 Anderson's Continuous-Media File System . . . 14
2.3 Multimedia Storage Manager and Rope Server . . . 15
2.4 The Lancaster Continuous Media Storage Server . . . 15
2.5 Cambridge Network Continuous Media Server . . . 16
2.6 Calliope . . . 16
2.7 User-Safe Disk . . . 17
2.8 Symphony . . . 18
2.9 The Tiger Video File Server . . . 19
2.10 Summary . . . 20

3 File-system traffic **21**
3.1 Continuous-media traffic . . . 22
3.1.1 Motion-JPEG . . . 23
3.1.2 Inter-block delays . . . 24
3.2 Best-effort traffic . . . 26
3.2.1 The Sprite File-System Traces . . . 29
3.2.2 The Hewlett-Packard Disk Traces . . . 30
3.3 Summary . . . 36

4 I/O technology **37**
4.1 I/O devices . . . 39
4.1.1 Disks . . . 39
4.1.2 Disk interfaces . . . 42
4.1.3 Networks . . . 43
4.1.4 Memory technology . . . 44
4.1.5 I/O bus technology . . . 45
4.2 A measurement environment . . . 49
4.3 Disk I/O measurements . . . 50
4.3.1 Zone layouts . . . 52
4.3.2 Seek measurements . . . 53
4.3.3 Single disk performance, disk cache enabled . . . 54
4.3.4 Single disk-read performance, disk-cache disabled . . . 55
4.3.5 Single disk-write performance, disk-cache disabled . . . 58
4.3.6 Multi-disk and PCI-bus performance . . . 59
4.3.7 Large PCI bursts . . . 64
4.4 Network performance . . . 68
4.5 Combined performance . . . 71
4.6 Summary . . . 72

5 Clockwise **73**
5.1 Environmental considerations . . . 75
5.2 Clockwise core . . . 76
5.2.1 Clockwise disk organization . . . 76
5.2.2 Using dynamic partitions . . . 78
5.2.3 Data-placement techniques . . . 78
5.2.4 Memory management policies . . . 79
5.2.5 Efficient I/O . . . 81
5.3 Best-effort applications . . . 82
5.4 Continuous-media applications . . . 85
5.4.1 QoS parameters . . . 87
5.4.2 AVA recordings . . . 88
5.4.3 AVA index information . . . 89
5.4.4 AVA stream synchronization . . . 91
5.5 Summary . . . 92

6 Disk scheduling **95**
6.1 Real-time scheduling techniques . . . 97
6.1.1 Liu and Layland . . . 98
6.1.2 Nonpreemptive (EDF) scheduling . . . 99
6.1.3 Real-time servers . . . 100
6.1.4 Other real-time work . . . 101
6.2 Clockwise disk scheduling . . . 103

6.3 Applicability . . . 107
6.3.1 Periods and service times . . . 108
6.3.2 Using ΔL . . . 109
6.3.3 Best-effort request selection . . . 109
6.4 Other approaches . . . 111
6.4.1 User-Safe Disk (USD) . . . 111
6.4.2 Latest Start Time (LST) scheduling . . . 112
6.5 Simulations . . . 113
6.5.1 Sequential layouts . . . 116
6.5.2 Random layouts . . . 121
6.5.3 ΔL scheduling and memory usage . . . 124
6.5.4 Best-effort QoS . . . 127
6.5.5 Quality of Service crosstalk prevention . . . 131
6.6 Summary . . . 133

7 Performance evaluation **135**
7.1 Validating simulated performance results . . . 136
7.2 Service-time predictions . . . 140
7.2.1 Read performance . . . 140
7.2.2 Clockwise disk caches . . . 144
7.2.3 Write performance . . . 146
7.2.4 Multi Quantum Atlas-II service times . . . 150
7.2.5 Multi Seagate-Cheetah service times . . . 151
7.2.6 Bad and error blocks . . . 153
7.3 QoS crosstalk prevention . . . 154
7.4 Combined network and disk performance . . . 155
7.5 CPU usage . . . 156
7.6 Summary . . . 157

8 Discussion **159**
8.1 Parallel disks . . . 160
8.2 Clockwise bandwidth considerations . . . 163
8.3 Clockwise for Linux . . . 164
8.3.1 Linux CPU scheduling . . . 165
8.3.2 Linux disk scheduling . . . 165
8.4 Service-times predictions . . . 166
8.5 Tertiary storage . . . 167
8.5.1 Demotion and promotion of data in Clockwise . . . 168
8.5.2 Archival support . . . 169
8.6 Summary . . . 169

9 Summary and conclusions **171**

References **174**

Acknowledgements **181**

Publications **183**

Post script **185**

Abstract

This thesis addresses the problem of implementing mixed-media storage systems. In this work a mixed-media file system is defined to be a system that stores both conventional (best-effort) file data and real-time continuous-media data.

Continuous-media data is usually bulky, and servers storing and retrieving many streams simultaneously require high throughput and capacity. To design a system that can support such bandwidth requirements, a thorough understanding is required of current I/O technology. An overview is given of commodity hardware that can be used for this purpose and performance experiments are presented that show how to use this hardware efficiently.

The measurements are used to design Clockwise, a mixed-media storage system. Clockwise is organized such that the bulk of the data flows between network and disk without requiring the CPU for transfers. Best-effort data is stored on the same set of disks as the continuous-media data.

Clockwise's task is to divide the server's physical memory and disk between best-effort and continuous-media applications. Memory needs to be managed explicitly, or else non-behaving applications can hog the memory in a way that other applications may loose continuous-media data. Disks need to be scheduled explicitly to guarantee real-time delivery of data on disk. Continuous-media applications use the real-time capabilities of Clockwise to guarantee smooth playback and record operations of digitized audio and video streams.

When best-effort requests are queued in the disk queue behind one or more (bulky) real-time requests, the latencies of best-effort requests can be severe. A new disk scheduler is presented that schedules best-effort requests before real-time requests when real-time request deadlines are guaranteed.

The capabilities of the disk scheduler are presented through simulations and real measurements on a Clockwise machine. It is shown that Clockwise can sustain a high best-effort load ($>$ 4,000 operations per minute) concurrently with a real-time load of up to 21 MB/s on an I/O architecture that is only capable of transferring data at a maximum rate of 25.3 MB/s. The Clockwise disk scheduler never misses a real-time deadline and schedules best-effort requests with reasonably low latencies.

*This work is funded by the European Union's ESPRIT BRA project 6586 (Pegasus-I) (1992 – 1995) and LTR project 21917 (Pegasus-II) (1996 – 1999).

Samenvatting

Dit proefschrift behandelt de implementatie van een *mixed-media* opslagsysteem. Een mixed-media opslagsysteem is een systeem dat zowel conventionele (*best-effort*) file-systeem data als *real-time*, continue media data kan opslaan.

Continue media data stromen zijn in het algemeen omvangrijk. Een opslagsysteem dat een (groot) aantal van dergelijke stromen tegelijk moet kunnen afspelen en opnemen, dient te kunnen omgaan met een grote data doorstroming. Voordat een dergelijk opslagsysteem ontworpen kan worden, is derhalve een studie naar de mogelijkheden van I/O apparatuur nodig. In dit proefschrift wordt een overzicht gegeven van standaard I/O apparatuur. Verder wordt een aantal *performance* experimenten beschreven waaruit blijkt hoe dergelijke hardware efficient gebruikt kan worden.

De meetresultaten zijn gebruikt voor het ontwerp van een mixed-media opslagsysteem. Dit systeem, Clockwise, kan zowel continue media data als *best-effort* data opslaan. Het zorgt ervoor dat continue media data tussen het netwerk en de disk interface(s) stroomt zonder dat de CPU nodig is voor het datatransport. Best-effort data wordt opgeslagen op dezelfde disks als de continue media data.

Clockwise verdeelt het fysieke geheugen en de disks tussen de *best-effort* en continue media applicaties. Het geheugen dient te worden verdeeld en beheerd zodat applicaties die zich niet gedragen, niet de kans krijgen andere applicaties dermate te beinvloeden dat deze continue media data moeten weggooien. Disks dienen geregeld te worden om *real-time* garanties te kunnen geven aan de overdracht van data op disk. Clockwise applicaties gebruiken deze *real-time* functionaliteit voor het opnemen en afspelen van gedigitaliseerde audio en video.

Wanneer *best-effort* disk operaties achter een serie van (omvangrijke) *real-time* disk operaties geregeld worden, lopen de *service* tijden van de *best-effort* operaties op. Clockwise gebruikt een nieuw type disk regelaar die het toestaat om *best-effort* operaties voor *real-time* operaties uit te kunnen voeren zonder dat de *deadlines* van de *real-time* operaties in gevaar worden gebracht.

Door middel van simulaties en metingen in een echt Clockwise systeem, is aangetoond dat de aanpak werkt. Clockwise kan een hoge *best-effort* werklast weerstaan (> 4000 operaties per minuut), tegelijk met een totale *real-time* werklast tot 21 MB/s op een systeem dat maximaal 25.3 MB/s kan verwerken. De Clockwise disk regelaar mist nooit een *real-time deadline* and regelt *best-effort* operaties met (redelijk) lage *service* tijden.

Chapter 1

Introduction

The advent of digital audio and video enables the storage, retrieval and manipulation of digitized audio and video in computer systems. Instead of dealing only with textual or numerical data, computer systems now need to deal with a totally different type of data: one that is *voluminous* and *isochronous*. This thesis addresses *how* to combine digitized audio and video storage with textual and numerical data storage.

To understand why digitized audio and video are referred to as *continuous media* it is important to understand what constitutes digitized audio and video. A/D and D/A converters produce or consume a continuous stream of digital samples of audio and video (hence: continuous media). An audio sample is usually represented by an integer that corresponds to the amplitude of the sound at an instant. An audio stream is constructed from a stream of these samples at a fixed sample rate (*e.g.* DAT quality audio contains 48,000 samples per second). A video sample represents the red, green and blue value, in some encoding, of a *pixel*. A video *frame* holds all pixels that can be seen through the lens of a camera. A video stream is constructed from consecutive frames at a fixed frame rate (*e.g.* PAL video is usually defined as 25 frames per second).[1] Although in reality digitized audio and video are not continuous but discrete, they are often referred to as continuous media because they consist of a continuous stream of discrete values.

Digitized audio and video are quite voluminous. The data throughput ranges from 8 KB/s for low quality audio to 112 MB/s for uncompressed HDTV-quality video.[2] An audio sample is usually an 8-bit integer for low end audio to two 16-bit samples for stereo high-quality audio. Professional use of digitized audio uses even more bits per audio sample. Given sample rates from 8 KHz to 48 KHz, audio bandwidth is approximately between 8 KB/s to 187.5 KB/s. A PAL video field holds 768×288 pixels and given 5 bit red, green and blue encoding at 50 fields per second, a PAL signal easily consumes 21 MB/s. Fortunately compression techniques remove redundancy from video streams and Motion-JPEG [102] streams, for example, can represent video of the same perceived quality in less than 1.5 MB/s.

A class of computer systems that should be able to manage continuous-media data well are

[1] PAL consists of 50 fields per second with 25 fields for the even scan lines, which are interlaced with 25 fields odd scan lines.

[2] Throughout this thesis KB represents 2^{10} bytes, MB represents 2^{20} bytes, GB represents 2^{30} bytes, Kb represents 2^{10} bits, Mb represents 2^{20} bits, and Gb represents 2^{30} bits.

real-time systems. These systems are systems that are primarily used to control physical processes in real time, such as controlling an engine. These systems must have a notion of the actual time at any given instant so that they can adjust the state of the physical process based on time.

Real-time applications require timing guarantees from the underlying real-time system. Not meeting a deadline means that a system error can occur and depending of the type of real-time system (hard or soft real-time), a true system error or a recoverable error occurs.

Real-time systems use schedulers that know the worst- (or average-) case behavior of each of the applications that need to run. Based on the timing information real-time systems perform a *schedulability test* to make sure that all applications can run without missing *deadlines*. This usually implies that real-time systems are not as efficient as non-real-time systems – a real-time system also needs to account for the worst-case scenario.

At first, it may seem logical to implement continuous-media systems through real-time techniques. There are, however, a few problems with this approach. Most real-time systems that exist today do not deal with 'real time' in the practical sense. Real-time systems first make sure that a set of applications is schedulable without missing deadlines by considering the resource demands. Once the schedulability of a set of applications is established, they convert timing constraints into priorities and schedule applications based on their priority. Even pure *Earliest Deadline First* (EDF) scheduling [66], which has a notion of time in its name, only uses the deadline of a task as an ordering principle.

Real-time systems usually use priorities to execute the real-time tasks as quickly as is possible to minimize the chance that a deadline is missed. However, a continuous-media system only requires the periodic execution of its tasks, which not necessarily implies that a task needs to run as quickly as is possible. Hence, it seems more appropriate to discuss real-time issues in terms of time rather than priorities.

Other problems with existing real-time techniques for continuous-media applications are that such systems need a measure for the worst-case behavior of the tasks. This information may not always be available for continuous-media applications since compressed data streams may require a variable service time to compress or decompress the continuous-media data. Also, not meeting a deadline in a continuous-media system is usually not a disaster: if such an event is noticed at all, all one notices is a short hiccup in a video or audio stream.

Continuous-media systems are often used by people communicating with other people or who are watching or listening to an earlier made audio or video recording. They would like to do this on their own workstation that also needs to run best-effort applications. Usually real-time systems are good in guaranteeing that the real-time constraints are met; they usually do not support best-effort applications well.

One can describe a mixed-media computer system as a system that combines the properties of a best-effort system and a real-time system. The best-effort support is required to run all of the user's best-effort tasks and the real-time support is required to make sure that all of the continuous-media tasks run on time.

The mixture of best-effort and real-time support is precisely what the ESPRIT Pegasus project addresses [100]. The Pegasus project has built a general purpose operating system that supports both best-effort and real-time applications that run in a single workstation. The Pegasus system is built from the ground up and involves the design and implementation of an operating-system

kernel that supports best-effort and continuous-media scheduling. This kernel is called Nemesis and is built by the University of Cambridge [64]. Also, a new network protocol stack is built with support for the explicit scheduling of continuous-media packets [11]. This work is extended by the Swedish Institute for Computer Science (SICS) to include a TCP/IP stack that can schedule IP packets explicitly. Lastly, a new file system is implemented that supports both best-effort and continuous-media traffic. Most of the work that is present in this thesis is part of the file-system portion of the Pegasus project.

1.1 Mixed-media file systems

The problem a building a *mixed-media file system* is the problem of combining best-effort file traffic and continuous-media traffic in a single storage architecture. This means that the system that implements the combined storage needs to be concerned about the division of computer resources among the continuous-media and best-effort tasks that make use of the service: best-effort tasks that require as-quickly-as-possible service and continuous-media tasks that require timely delivery of data.

Three aspects are important for the storage and retrieval of continuous media: the aggregate *bandwidth* of the service, the available storage *capacity* and the *timely* delivery of continuous-media data. Given the timeliness and high bandwidth of the data, it is a challenge to design a storage service that can guarantee the timely recording and playback of a (possibly large) number of continuous-media streams simultaneously. The amount of data that is involved when storing continuous-media data implies that a continuous-media system needs to be able to manage vast amounts of storage space.

High bandwidth service implies that the system needs to be able to transfer data efficiently between source and sink. To allow efficient data transfers through a storage system, such a system needs to be able to cope with large disk blocks to optimize data transfers and the CPU must not be in the data path when transferring data [73, 101].

Timeliness for continuous-media traffic means that each (audio or video) frame needs to be processed within certain time limits. A 25 frames per second PAL video stream, for example, requires a new frame every 40 ms. When frames do not arrive at precisely the correct pace, the *inter-frame delay* is not constant: the deviation is called *jitter*. The human ear and eye can tolerate some jitter, but there are limits to this amount before it becomes noticeable.

The timeliness and bandwidth requirements of a stream are collectively called the *Quality of Service* (QoS) parameters of a continuous-media stream. When a stream starts, a client presents its QoS parameters to the mixed-media file system, which, on its turn, uses these parameters to determine if it can accept the new request without violating QoS guarantees of other already admitted streams.

The QoS parameters are used to established a scheduling contract between the client and the server. If a client application behaves according to its negotiated QoS parameters, the file system must guarantee service according to these parameters. If a client application does not behave according to the QoS parameters, the file system does not guarantee that it can meet the application's demands. All it can do in this case is to do its best, which means that the guaranteed

service degrades to a *best-effort* class of service.

When an application does not behave according to its negotiated QoS parameters but uses more resources (*i.e.*, it *'misbehaves'*), the application may impede on the resources that are meant for other *'behaving'* streams. For example, if a recording application temporarily uses more bandwidth than it has originally requested it may use resources that are meant for other client applications. The system needs to schedule its resources such that the 'misbehaving' application cannot impede on the resources that are meant for the 'behaving' applications. When a 'misbehaving' application disrupts the service of a 'behaving' application, it is said that there is *QoS crosstalk* [9]. QoS crosstalk prevention is the technique to prevent such disruptions.

Summarizing, to support continuous-media data storage in a mixed-media storage system, a server:

- Implements an efficient and high-bandwidth data path between network and storage devices.
- Offers QoS to the clients.
- Prevents QoS crosstalk between the various clients that make use of the same storage facilities.

'Ordinary' or *best-effort* traffic is file-system traffic that originates from (interactive) applications (possibly) running on remote workstations. Examples of these applications are text editors, compilers, news readers, and Web servers. To understand how to optimize for such traffic, a number of studies to the nature of this traffic are performed. These best-effort applications generate bursty traffic [74, 6, 28, 87] that is also characterized by a high overwrite factor [13]. For this work special emphasis is placed on UNIX best-effort file traffic, which is commonly referred to as scientific best-effort data.

I/O throughput and request latency are important issues for best-effort file traffic.[3] Throughput determines how many operations can be executed per time quantum, latency determines the responsiveness of a system, *i.e.* the quicker the system services best-effort requests, the better service it provides to the client applications. Since the applications are usually interactive, it is likely that a user is waiting for the operation to complete. When requests complete sooner, the user waits less.

Achieving low I/O latencies implies that the disk should not be in the data path. If the user needs to wait for the data to get to or from disk, the operation can easily take a few tens of milliseconds when the request is serviced directly. Worse, in a mixed-media file system the best-effort data request can be queued behind large continuous-media data requests and can be delayed substantially longer. When data is serviced directly from the client or server file-system cache, only file-system protocol overhead, possibly extended with a network delay, contribute to the latency.

If best-effort requests cannot be serviced from the file-system cache, the blocks need to be served from secondary storage. A mixed-media file system needs to be able to give precedence to best-effort requests despite queued continuous-media requests or else best-effort requests can

[3] It goes without saying that the stored information also needs to be *persistent*.

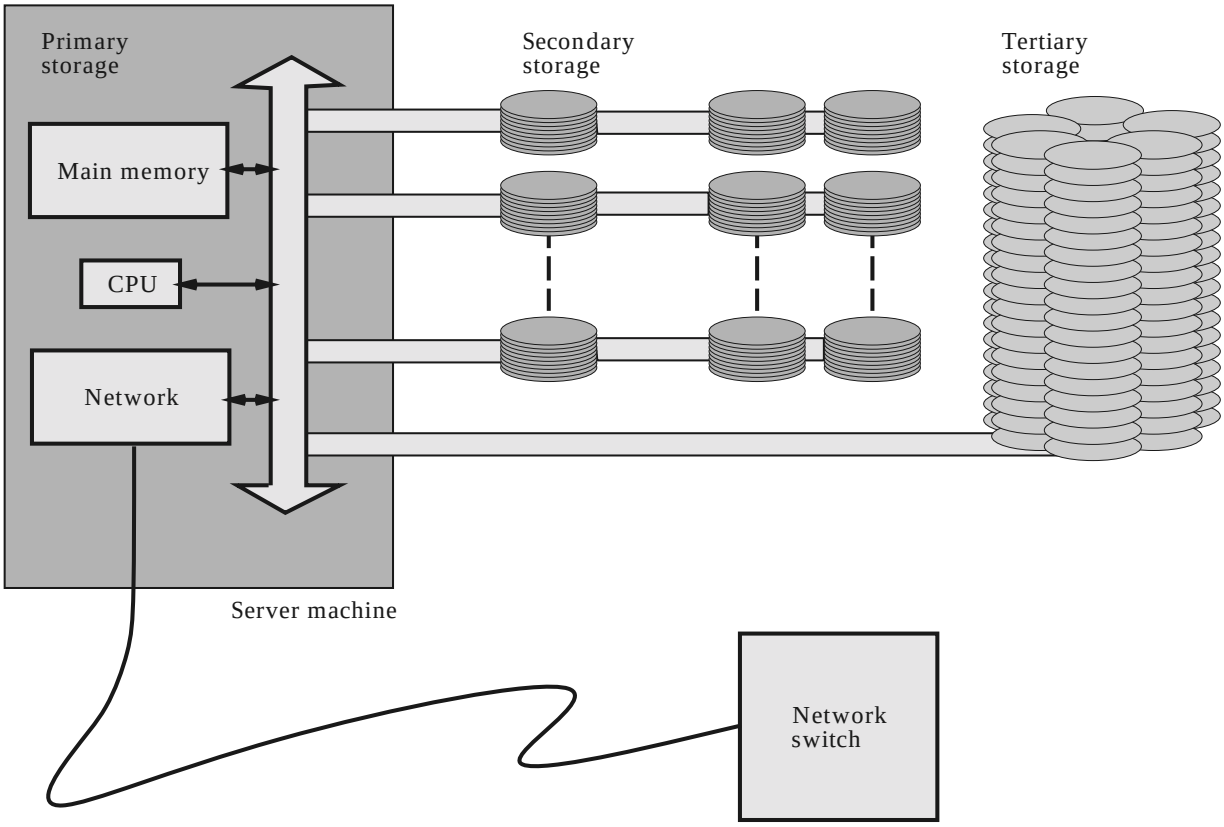

Figure 1.1: A mixed-media file system hardware organization.

be substantially delayed. The Symphony system [93], for example, calculates a *Latest Start Time* (LST) schedule for continuous-media (real-time) requests and allows best-effort requests to be serviced first when the latest start time of continuous-media request are not violated.

A mixed-media file system combines best-effort (UNIX file-system) storage with continuous-media data storage. It needs to be concerned with dividing CPU, memory capacity, memory bandwidth, network bandwidth, storage capacity and storage bandwidth over the best-effort and continuous-media real-time applications. Figure 1.1 presents a typical hardware architecture for a mixed-media file system. Since the mixed-media file system stores large continuous-media files, it needs to be able to manage a large quantity of storage capacity. For economical reasons, the storage capacity consists of a hierarchy of storage devices. Main (or primary) memory storage is used to cache best-effort file-system data and to buffer continuous-media data. Data storage in main memory has a short life-span: once continuous-media traffic is sent over the network, or is written to disk, it is discarded from main memory. Best-effort file data is only cached or stashed in main memory: once the memory is required for other purposes, the data is simply discarded or flushed to secondary storage.

Secondary storage is used for two purposes in a mixed-media file system: it stores the best-effort file systems and it provides medium-term storage facilities for continuous-media files.

Since the best-effort file systems usually undergo many changes throughout their lifetime, such file systems need to reside on a storage medium that, on one hand provides reliable storage, and on the other hand allows frequent updates. Also, the medium needs to be fast since it is not likely that the entire working set of all best-effort clients fit in the server's main memory: some of the data requests need to be read from secondary storage before it can be sent to the clients. Standard disks provide exactly this functionality so that this medium, possibly organized in an array to improve performance and resilience against faults, implements long-term storage facilities for best-effort file systems.

Continuous-media files are only buffered on secondary storage because secondary storage is expensive. When compared to standard analogue video tapes or tertiary storage media such as digital tapes, ZIP drives, CDs or DVDs, disks are an order magnitude more expensive to store continuous-media data. So, as long as continuous-media files are recorded for the first time, are edited, or need to be available for massive parallel playback, they are stored on secondary storage. Once a continuous-media file is not frequently accessed or is not altered anymore the data migrates to tertiary storage. Wilkes *et al.* [104] have called this *demotion* of data.

Tertiary storage provides archival storage facilities for best-effort file systems and provides long-term storage facilities for continuous-media files. Daily backups of best-effort file systems take place on secondary storage and file-system *deltas* are written to tertiary storage: all of the updates that are made throughout the day are copied to, for example, a CD. The backups can be made available in the best-effort file system's secondary-storage name space to allow users to access old files by themselves, much like the Plan 9 tertiary-storage facilities [77]. Continuous-media files are stored on tertiary storage once they are not used frequently anymore to free up expensive secondary storage. Playback from tertiary storage is usually still possible by either caching the data on secondary storage again (*promotion* of data [104]), or by directly sending the data from the tertiary-storage devices onto the network, depending on the expected usage pattern of the requested continuous-media file.

It is expected that the main memory capacity range up to 512 MB of RAM. Secondary storage space is expected to be at least 100 GB large and tertiary storage is expected to hold at least 1 TB worth of data.

The problem of designing a good mixed-media file server can best be characterized as follows: it is the mixed-media file server's task to divide primary, secondary and tertiary storage resources to the clients (of several tens) of best-effort file systems, which demand quick delivery of data *and* at it needs to guarantee *timely* service to a significant number of continuous-media storage applications simultaneously.

There are three operating-system entities involved in the sharing of resources in a mixed-media file system: the operating-system kernel, the mixed-media file server and the network drivers.

A continuous-media operating-system kernel makes sure that the division of the CPU to the applications that run on the machine is done such that continuous-media applications run 'on time.' If, for example, a PAL video is played back, the application that transmits the packets needs to be started every 40 ms. The operating-system kernel needs to schedule such tasks every 40 ms.

If the operating-system kernel cannot, to some extent, guarantee such a real-time behavior, it

is unsuitable as a basis for a mixed-media file system. The various versions of the popular UNIX operating systems [82], for example, are totally unsuitable as a basis for a mixed-media file system: UNIX cannot guarantee a periodic service. Nemesis [64], the Pegasus operating-system kernel, however, allows the explicit scheduling of tasks. Every application presents CPU *Quality of Service* (QoS) parameters in terms of *period*, *slice* and *latency* to the operating-system kernel. The period parameter is determined by the periodicity of the application. The slice parameter determines how long the application requires per period. The latency parameter determines how quickly the CPU must be acquired within a period. When Nemesis accepts the QoS request, it guarantees that an application can use the CPU periodically as specified in the QoS contract. An operating-system kernel such as Nemesis is ideal basis for a mixed-media file system.

Memory management is a shared responsibility of the operating-system kernel and mixed-media file system. Continuous-media applications require memory for temporary storage of continuous-media before it is transmitted on a network or to a disk. Best-effort applications such as UNIX file systems, require memory for temporary storage of dirty (*i.e.*, data that is not yet backed by secondary storage) and non-dirty file blocks. In order to read or write data to or from a disk, an application requires (physical) memory. If the availability of memory is not guaranteed, it is of no use to guarantee disk bandwidth. For convenience and efficiency, memory management is divided between the operating-system kernel and the mixed-media file system: the operating-system kernel implements the mechanism, the mixed-media file system implements the memory distribution policy.

A mixed-media file system allocates the storage bandwidth and capacity to the storage applications. Each of these storage applications can have a different storage requirement: continuous-media applications can be characterized by their periodic I/O requests, while best-effort applications are only interested in low latency disk operations. Storage capacity is divided upon availability amongst the storage applications.

A mixed-media file system needs a network to receive or transmit continuous-media or best-effort data. There exists a wide variety of network technologies, each with its own peculiarities. A popular network today (and has been for quite some time) is Ethernet. This shared bus was originally designed as a 3×10^6b/s network and is now widely available in a 100×10^6b/s version. Unfortunately, when the bus is shared without centralized control, its use for continuous-media data transmission is limited because one transmitter can easily disrupt the timely service of another system. Instead, networks like ATM can schedule data through the network by associating a network QoS setting to a connection between a source and a sink. When hardware support is absent, a network device driver can enforce some network QoS [11].

As said, this thesis mainly focuses on mixed-media file storage. This does not imply that the other areas are not addressed: as is also shown in this thesis, only a close collaboration between all components inside such a system makes it possible to create a flexible and efficient mixed-media file system.

1.2 Scope of this thesis

This thesis addresses *how* to construct a mixed-media file system that is capable of providing storage facilities for real-time and best-effort applications in a flexible and efficient manner. The real-time functionality is primarily meant to service continuous-media applications. This, however, does not exclude other real-time uses of the system. The best-effort functionality is meant to service UNIX file-system applications. The following issues are addressed:

- In order to store continuous-media data and best-effort data, a thorough understanding of these types of data is needed. A study is performed to the nature of continuous-media and best-effort file traffic.

- The limiting factor in continuous-media file systems is often throughput. An analysis of the available hardware is required to understand how to build a service that can cope with high-data rates.

- What is the design of a mixed-media file system that can support both high-bandwidth data traffic *and* best-effort file traffic in the same architecture? This includes a discussion how to layout best-effort and high-bandwidth data on disk and it includes a discussion how to organize the mixed-media file system to allow for efficient bulk data transfers through through the server's machinery.

- What are good policies to distribute memory resources to each of the storage applications from both storage classes so that continuous-media tasks can use *enough* memory resources to deal with constant bit rate and variable bit rate audio and video streams and best-effort applications can allocate as much memory as is possible to improve best-effort latencies through caching techniques?

- Current (disk) scheduling techniques cannot cope with un-admitted (best-effort disk) load in a real-time schedule. How can such un-admitted best-effort load be inserted in a real-time schedule, so that best-effort request latency is minimized and real-time request deadlines are met?

- *QoS crosstalk* occurs when 'misbehaving' applications use resources that are meant for other 'behaving' applications. How can QoS crosstalk be prevented in a mixed-media file system?

1.3 Rationale

This thesis presents a number of techniques that are already (well) known in the field of computer science. In particular the way the mixed-media server addresses the problem of efficiency and how to prevent QoS crosstalk in a system is known. However, today there does not exist a single system that combines all of the issues that are described in Section 1.2 into a single architecture.

Fundamentally new is the way disks are scheduled in the mixed-media file system. This new disk scheduler pre-calculates schedule slack time (*i.e.*, unallocated schedule time), which is used to prioritize best-effort traffic and to accommodate for continuous-media streams that require more resources compared to their negotiated QoS contract. To date, mixed-media file systems had to rely on the computation of so-called *Just In Time* schedules to compute how much slack time is available. This *Just In Time* schedule is computationally demanding, it needs to be calculated online, it can lead to poorly scheduled disks and, as is shown in this thesis, scheduling best-effort requests in a *Just In Time* manner can lead to missed real-time deadlines. In fact, the slack time that is calculated by the new disk scheduler is a fundamental property of all nonpreemptive schedulers: without it, it is impossible to schedule unadmitted load without endangering real-time deadlines.

To demonstrate that the scheduling approach that is presented in this thesis works, it is integrated in a new mixed-media file system. The entire system is entirely built in the programming language C and consists of slightly more than 100,000 lines of C-code, which includes the code for the recording and playback of audio and video files and an entire UNIX file system.

Where possible, code is re-used from other projects. In particular, the UNIX file system support is entirely based on the NetBSD file-system implementation. This has the advantage that there are no worries about the stability of the system. Secondly, the entire device-driver code is borrowed from the Linux system.

1.4 Road map of this thesis

Chapter 2 presents related work. In this chapter a number of other systems are reviewed and their strengths and weaknesses are presented.

Chapter 3 describes the nature of continuous-media and best-effort file traffic in detail.

Chapter 4 presents current I/O technology and raw I/O measurements for a modern computer system. The reason for performing these measurements is to learn how a mixed-media file system must be designed to maximize user I/O bandwidth. In particular, this chapter shows that an off-the-shelf PC can be used as an I/O machine with an aggregate I/O performance in excess of 100 MB/s. 'Don't hide power' [58] is the key issue for designing high-bandwidth file systems.

Chapter 5 presents *Clockwise*, a mixed-media file system that is designed to service both real-time storage applications, in the form of continuous-media applications, and best-effort applications in the form of UNIX file systems.

Chapter 6 deals with an important aspect of Clockwise: how to schedule disks so that deadlines of continuous-media traffic are always met, while trying to give precedence to best-effort traffic. A new disk scheduler is presented and its performance is measured by means of discrete-event simulations. The performance of alternative schedulers is also measured and compared to the new disk scheduler. The simulator that is used for the experimentation uses the performance characteristics that are described in Chapter 4.

Chapter 7 presents real performance measurements with Clockwise. Short periods from the discrete-event simulations are re-executed on a real Clockwise to validate simulated performance results. Also, this chapter shows the actual performance of requests that are executed on the

mixed-media file system in relation to what is measured in Chapter 4.

Chapter 8 contains a discussion on related file-system research and possibilities to extend the Clockwise work. When the throughput of a single Clockwise server is not enough to satisfy an application's need, multiple Clockwise servers need to be used in parallel. This chapter presents how to extend the current Clockwise to support distributed operations. Also, as is shown in this thesis, current PC hardware cannot be used efficiently and when there is a substantial load on a PC its performance becomes unpredictable. This chapter presents an overview how to change hardware to support more efficient hardware scheduling. In fact, changing the hardware is a pre-condition for building a real-time system on such general purpose hardware. Chapter 8 also describes an approach to integrate Clockwise technology into mainstream operating systems, it describes a method for finding disk scheduling parameters semi-automatically, and it describes how Clockwise can support tertiary storage.

Finally, Chapter 9 summarizes this thesis and presents the conclusions that can be drawn from this work.

Chapter 2

Related work

There exist storage systems that optimize for best-effort file traffic, continuous-media file traffic or both. A large number of the best-effort systems are only capable of storing best-effort data. These systems do not have a notion of real-time constraints and their internal scheduling techniques only try to optimize for best-effort latencies. An example of these types of systems are the UNIX file systems.

Another class of existing storage systems only deals with the storage and retrieval of continuous-media data. These systems make sure that many concurrent continuous-media streams can use the storage system at the same time without missing deadlines. This category of systems includes *Video On Demand* (VOD) systems where subscribers can order a movie that is played back on a local display.

There also exists a class of mixed-media file systems that (try to) accommodate best-effort as well as continuous-media file traffic. These systems store both kinds of data on the same set of disks, and try to schedule disk requests so that the continuous-media applications can rely on throughput guarantees and best-effort applications are served as quickly as is possible. There are not many of these mixed-media file systems.

2.1 Existing UNIX file systems

UNIX file systems are primarily used for, what is called here, best-effort data storage. These file systems optimize for the expected UNIX file traffic. Many of these optimization policies are in conflict with what is required for continuous-media file traffic. In particular, the layout of data on disk, the caching principles and disk-scheduling techniques are inadequate for continuous-media file storage. These systems are capable of serving UNIX files well: they are good at serving small files and can sustain bursty loads.

Three components from UNIX file systems determine how well suited a UNIX file system is for mixed-media data storage: the way caches are structured, meta-data management and the data layout on disk.

2.1.1 UNIX *Least Recently Used* (LRU) caches

All of the UNIX file systems employ a disk-block cache from which to satisfy read requests. The read cache is usually implemented with a LRU replacement policy. This cache holds all of the recently accessed file blocks with the hope that these blocks are re-accessed again.

When a continuous-media file is played through a LRU based block cache, the LRU cache policy replaces best-effort file blocks by those from the continuous-media file. It is likely that after playback of a continuous-media file, a significant part, if not all, of the disk cache is cluttered up with continuous-media disk blocks. Since continuous-media files are usually played sequentially from start to end, caching a continuous-media file in an LRU cache only makes sense when the entire movie can be cached. It is likely that for large continuous-media files or when a number of continuous-media files are played back simultaneously, the beginning of the continuous-media files is no longer cached by the time playback ends so that the entire continuous-media file needs to be re-read for another playback. In any case most UNIX file blocks are evicted from the read cache and the cache is filled with data blocks that are hardly worth caching.

Most UNIX file systems buffer newly written data in memory before it is flushed to disk. This is done in the hope that the file will be quickly overwritten with a new version or is entirely deleted from the file system, which happens regularly in UNIX file systems [87, 13]. If data is overwritten or deleted before it is sent to disk, most UNIX file systems do not update the disk with the old version. After the dirty data is flushed to disk, the flushed block is stored in the read cache for later reference. Most UNIX systems flush data every 30 seconds to disk through the update daemon.

The UNIX disk update policy is inadequate for continuous-media file storage. If a number of continuous-media streams are recorded simultaneously, it is possible that the entire cache space is filled up with unwritten continuous-media data blocks before the update timer goes off and starts flushing data. When the cache is full, new continuous-media blocks can therefore only be stored in the dirty cache when other blocks are first written to disk. The flushing process is slow, and the lack of cache space may cause discarding of new incoming continuous-media data. After continuous-media data is flushed to disk, the flushed cache blocks are added to the read cache, which leads to the earlier described read cache problems.

An application can write data synchronously to disk, so that data is not buffered in the dirty-block cache. However, the written data is still stored in the read cache.

2.1.2 UNIX inodes

A UNIX inode is a data structure that describes which disk blocks are allocated to a UNIX file. The inode is a data structure that is kept in a well known location on disk and, when a file is opened, is copied to memory. The addresses of the first hundred (or so) allocated disk blocks are stored in the inode data structures itself. Since inodes have a fixed size, extension blocks are used to address disk blocks for files that are larger than what can be described by a single inode block. These extension blocks are called the indirect and double indirect blocks. The indirect blocks refer to disk blocks that refer to the actual data blocks. The double indirect blocks add yet another level of indirection. The disk address of the indirect block is found through the last

entries in an inode block, the double indirect block is found through the last entries of the indirect blocks.

Since continuous-media files are likely to be large, and since many UNIX file systems do not read in all of the indirect, double indirect address blocks when a file is opened, playing back a continuous-media file from a UNIX file system implies that these address blocks need to be read on demand. Since most UNIX file systems keep the block administration on distinctive locations on disk, the disk needs execute expensive seek operations to read in a few address blocks. This policy reduces the effective bandwidth of a disk substantially.

To record a continuous-media file, the situation is even worse. Since most UNIX file systems require block allocation to be executed in a predefined order with regular disk updates to maintain file-system consistency, the available disk bandwidth is reduced substantially.

2.1.3 UNIX file-system layouts

Different UNIX file-system disk layouts optimize different characteristics in UNIX file traffic. FFS [71] optimizes disk requests by organizing the file system so that related files, *i.e.* files that are located in the same directory, are stored close to each other in *cylinder groups*. When a UNIX file system is searched, both the directory and the files that are stored in the directory are needed. By arranging these directory and files in the same cylinder group, the disks need to seek less and the I/O latencies are reduced.

When continuous-media files are stored on a FFS UNIX file system, it is likely that all of the continuous-media file data is stored in a few cylinder groups. Data within a cylinder group can be read and written quickly since the disk arm does not need to be repositioned over long distances, which is advantageous to obtain a high I/O bandwidth. However, FFS does not allow applications to enforce this type of cylinder-group assignment to continuous-media files and combined with the aforementioned inode problems, the maximum enforceable utilization on a disk is likely to be low. Also, FFS cannot guarantee disk throughput, so that a continuous-media application can never be certain that blocks are delivered in time.

EFS [91] is an FFS with variably sized blocks called *extents*. EFS tries to store large files sequentially on disk to improve the sequential read and write performance. While EFS has improved on the efficiency of disk usage for large transfers, EFS still does not guarantee disk throughput.

VXFS [98] is an extend-based file system much like EFS. Although the layout of the VXFS file system is suitable for continuous-media data storage, VXFS cannot, just like the other UNIX file systems guarantee throughput. VXFS has shown that sequential data storage is beneficial for file-system throughput: a specially configured SUN workstation is able to deliver data at a sustained rate of 960 MB/s.

LFS [85] is a log-structured file system that has improved the write performance of the UNIX file system: batches of file-system updates (data and meta-data) are appended to the file-system log without many seek operations. By using a large (LRU) read cache, most read requests hit in the block cache, so that the disk is predominantly used for write operations. Overall, an LFS spends less time seeking compared to other UNIX file systems.

A standard LFS is unsuitable for continuous-media file storage. The reason for this is that a standard LFS does not guarantee disk throughput. Also, applications that use a standard LFS cannot enforce that blocks are allocated sequentially to optimize the available disk bandwidth. In fact, when two or more media streams are recorded simultaneously, it is quite likely that the blocks are stored interleaved on disk [67].

With effort, an LFS can be made to support continuous-media traffic. The file-system log is implemented by linking large *segments* of disk blocks. To enlarge the log with a new segment, it is allocated from a free segment list and the last segment's forward link is made pointing the newly allocated segment. By organizing the LFS in such a manner that continuous-media data is stored in private *continuous-media segments* and all meta data that is needed to access the continuous-media data is stored in the standard LFS file-system log, the bulky continuous-media data can be efficiently read from and written to disk.

However, building such a combined service on a LFS is still a major task: all of the other UNIX file-system problems still need to be solved.

2.2 Anderson's Continuous-Media File System

Anderson *et al.* [2] describe a file system for continuous-media that makes use of sessions. Each client 'opens' a session with the continuous-media server and when the server accepts the session, the client is guaranteed a minimal performance from the server. Each session contains a FIFO that acts as an intermediary between the client and the server. The server automatically fills or empties the FIFO.

The server accepts new sessions if it has enough resources to schedule the new request through a *static schedule*. This means that, in principle, only the fixed-sized transfer rate of the stream is taken into account. To deal with variable bit rate streams, Anderson's continuous-media server makes use of a cushion that may be different for each stream. Since the record-rate requirements of streams may not be known in advance, there is a possibility that the cushion is too small. Each session is only given a fixed transfer buffer, which means that in the case that there is an overflow, data needs to be discarded – the continuous-media server does not try to allocate more buffer space dynamically.

With respect to disk-scheduling policies, Anderson *et al.* have analyzed three scheduling policies: static/minimal, greedy, and cyclical plan. The static/minimal scheduling policy uses the rate parameter to each session to determine an overall schedule. The greedy policy transfers as many bytes as it can for each session and the cyclical-plan policy tries to distribute slack time to the *bottleneck* session. Of the three scheduling policies, only the simplest one cannot deal with variable bit rate transfers. The other two policies distribute slack time to streams that need extra bandwidth.

Anderson *et al.* do not use an EDF scheduler because 'Policies for real-time CPU scheduling, such as earliest-deadline-first, are not immediately relevant because of seeks' *([2], p. 325)*. It is true that it is not efficient to preempt a disk transfer if a higher priority request arrives (if such an operation can be performed at all), but this can be solved by not allowing disk preemption and using EDF only as a disk-queue ordering policy.

Anderson *et al.* do not try to optimize disk I/O performance. They can configure their system so that it allocates large blocks on disk, but the disk scheduler tries to fill the FIFOs whenever it can. This means that, if only one block is available in the session FIFO, only one block is transferred. In that case the disk performance is reduced.

2.3 Multimedia Storage Manager and Rope Server

Rangan *et al.* [80] have designed and implemented a continuous-media file system that organizes continuous-media files in the form of *multimedia strands*, an immutable sequence of storage blocks, and *ropes*, a collection of independent strands. During the recording of a continuous-media session the system derives storage parameters such as the storage granularity (block size) and scattering value (separation between successive blocks). The continuous-media system has an admission algorithm that determines whether the storage or retrieval of a new strand can be admitted by the server without violating disrupting the playback or recording of other already admitted strands.

Rangan *et al.* claim that direct transfers from disk to display are preferable to transferring the data through memory as this '...requires double the internal bus bandwidth.' *([80], pg. 86)*. This is certainly true as is shown later in this thesis. However, when direct transfers are used, the sender of the data needs to prepare the data in a way that 'fits' the receiving video card, or the receiving network controller or video card requires functionality to alter the received continuous-media data to match the data format of the video card.

The continuous-media system that is described is not designed to store variable bit rate streams. When a variable bit rate stream is recorded one cannot tell in advance what the transfer rates will be so determining the storage parameters may proof difficult.

The storage granularity is used to lay out continuous-media blocks on the disk. It is not known what the effects are on the system when the storage devices become full. Finding empty blocks within the granule may be a difficult operation. It seems that the assumption is that there is always enough free space available.

2.4 The Lancaster Continuous Media Storage Server

Lougher [67] describes the Lancaster Continuous-Media Storage Server (CMSS), which is a specialized continuous-media server that is used in a small environment. The specialized server is based on Transputer technology and connects to a high speed FDDI network. Clients can access the continuous-media server through the FDDI network to retrieve and record their continuous-media files. Each client machine is equipped with a continuous-media box, which is able to digitize audio and video and send it through the FDDI network to a remote server. The client and server cannot compress the continuous-media data.

The CMSS uses a log-structured file system as the basis for the disk layout. The reason for this is that if only a single file is recorded, the data blocks for the file are stored contiguously on disk. This means that the system does not need to waste time seeking the disk when the file is

played back. If multiple files are recorded concurrently, the files are probably related, and will probably be played back concurrently as well. Since the data is stored in a log-structured fashion, blocks belonging to the concurrently recorded files are stored interleaved on the disk.

The CMSS uses an *admission test* that verifies whether new streams violate the continuity requirements of already admitted streams. Disk requests are ordered in cycles. A stream is admitted if there is enough slack time in the cycle for the new request. If so, the new request is added to the schedule. Since the cycle time is determined by the most consuming stream, streams that are less demanding have the problem that they can be allocated too many slots. This leads to a low disk utilization.

New streams can also be allocated in the *best-effort* stream class. In this class disk slack time is handed out in deadline order. However, no guarantees are given in this traffic class and it is unclear how this traffic class behaves.

2.5 Cambridge Network Continuous Media Server

Jardetzky [52] describes a specialized continuous-media server that is able to store and playback audio and video from and to a Pandora's Box. A Pandora's Box is a device that is located between an ATM network on one side and a video display, camera and microphone on the other side. It sends and receives ATM network packets with digitized audio and video, it renders the received video on the display and outputs the received audio on a set of speakers.

The continuous-media server consists of a (simple) ATM network interface, an ordinary CPU, some memory and one or more disks. The primary goal of the Pandora work is to demonstrate that a continuous-media server requires a specialized design.

The continuous-media server is an extent-based file system. The basic disk allocation unit is an extent of 90 4 KB blocks for video and 24 4 KB blocks for audio. These sizes are chosen such that the expected median file size (24 seconds) fits in one extent. Each allocated 4 KB block is recorded in the file's inode, which has a structure similar to the ordinary UNIX inode.

Jardetzky's continuous-media server's storage I/O systems outperforms the network, which means that the system can use a simple disk scheduling technique and relatively small blocks. In the experiments, the continuous-media server is able to process data from and to the network at a maximum rate of 3.2 Mb/s.

2.6 Calliope

Calliope [101] is a straightforward continuous-media server that can handle both variable and constant bit rate video. Clients preallocate the required bandwidth for a stream, which is translated by Calliope into a disk schedule. This disk schedule is maintained by a *duty cycle*. Requests from the duty cycle are continuously executed, which fill preallocated transfer buffers. More demanding streams are allocated more slots in the duty cycle. Variable bit rate streams need to allocate close to their peak bandwidth worth of slots in the duty cycle. Calliope always allocates

two buffers per stream for double buffering: while the disk subsystem is transferring one buffer, the client can fill or empty the other buffer. Calliope runs as a user process on FreeBSD.

The problem with Calliope is that when variable bit rate streams are transferred, the system needs to allocate bandwidth close to the peak bandwidth of the stream. The reason for this is that the system uses a static schedule and a strict double-buffering scheme, which implies that Calliope cannot anticipate on extra bandwidth demands.

A second problem with Calliope is poor performance. Calliope can at most transfer data at a rate of 4.7 MB/s on a Pentium PC with an FDDI network, which is not really satisfactory given today's disk speeds. Calliope cannot perform at higher rates because the system on which it runs can only handle a single SCSI interface card. In any case, since the CPU is in the data path for continuous-media data transmission, it is expected that Calliope cannot send more than 7.5 MB/s through a Pentium based PC.

2.7 User-Safe Disk

Barham's *User-Safe Disk* (USD) [10, 9] partitions the available disk bandwidth to a number of data streams that use the disk. Each stream requests a certain *Quality of Service* (QoS) from the disk, which, when the request succeeds, 'hands' the disk bandwidth to the stream. The novel approach by Barham is that each application is given a data flow to or from an USD disk with which it can do whatever it wants. When multiple streams use USD concurrently, USD schedules the requests through an *Earliest Deadline First* (EDF) scheduler [66].

USD is not a continuous-media file server. Its only task is to present the disk to the applications in such a manner that its behavior becomes predictable. A continuous-media file server can use a USD: for every stream the continuous-media server admits, a private USD stream is allocated. In fact, the $\mathcal{EFS}$ file system is an extent-based file system with continuous-media storage facilities that uses USD as back-end.

The USD EDF scheduler is based on the Nemesis Atropos CPU scheduler [83]. Every task allocates a slice C that can be used every period T. The EDF schedulability test makes sure that the total load is less than or equal to one:

$$\sum_{j=1}^{n} \frac{C_j}{T_j} \leq 1$$

The fundamental problem with this approach is disks are not preemptable while the Atropos schedulability tests assumes that tasks can be preempted. This means that it is unclear how the USD can guarantee QoS since USD can admit streams that are in reality unschedulable. When a requests misses a deadline, it is 'punished' by temporarily reducing the guaranteed service time of the task that issued the request [31]. As is shown by Sha *et al.* [92], when an EDF scheduled resource is overloaded, an arbitrary task may miss a deadline. This implies that USD may punish the wrong task. Section 6.4.1 presents the USD's scheduling problems in more detail.

When USD idles, the disk is 'given' to a stream that can use extra time. Although this policy makes sure that 'runnable' tasks do not have to wait for their next release time, it does not

prioritize best-effort traffic. Best-effort traffic is only scheduled when there is no real-time traffic available.

2.8 Symphony

The Symphony file system [93] is a mixed-media file system that is capable of storing continuous-media and best-effort data on the same set of disks. All of Symphony's disks together form a single logical disk. Clients can use the single disk abstraction or find out how their data is laid out on the disks: if clients need access to single physical disks, they can address the disks independently.

Symphony has addressed the problem of mixed-media disk scheduling. The Cello disk scheduler [94] is a two level disk scheduler, where the main class-independent scheduler employs a *First Come First Serve* (FCFS) scheduling policy and a number of class dependent disk schedulers schedule requests according to the application's needs. Class schedulers exist, which order requests for best-effort traffic, optimize for periodic/aperiodic traffic and for throughput intensive applications.

Requests move from the class dependent queues to the class-independent queue in several ways. The periodic and aperiodic requests move to the class independent queue in a *Just In Time* (JIT) manner: requests are first ordered in SCAN-EDF order [81] in the periodic/aperiodic request queues and are then moved to the class independent queue at their latest start time.

The class dependent best-effort scheduler uses a slack time stealing policy to find the earliest execution time for the best-effort request. This scheduler inserts the best-effort request into the class-independent queue whenever it finds slack time in the class independent queue. Slack time is identified when the disk idles or when a real-time request can be postponed for the duration of the best-effort request without missing a deadline.

The problem with Symphony's disk scheduler is that it is not apparent that a task set remains schedulable when best-effort traffic is inserted before the real-time requests when slack time is detected. In fact, as is shown later in this thesis executing requests through the *Latest Start Time* (LST) scheduling technique without considering minimal schedule-slack time leads to missed deadlines for real-time traffic. It is important to analyze the effects of inserting best-effort traffic into nonpreemptive schedules.

Symphony allows separate application classes to implement their own memory-caching policy. A central buffer manager is able to access all of the caches and to revoke buffers from the cache when they are needed somewhere else. For this, each class is required to maintain a *Time To Reaccess* (TTR) metric, which is used by the central buffer manager to determine the location of the cache buffer in its lists.

The caching policies implies that Symphony implements a central buffering policy and class dependent storage applications do not have full control over their own cache buffers. If buffers are required by the central buffer manager, it is this central buffer manager that decides which blocks are evicted and possibly flushed to disk instead of the class dependent policies.

Symphony runs as a single UNIX process. This means that it is not possible to enable account resource usage (*e.g.* CPU usage) on a per-client or per-class basis as is done in the Pegasus sys-

tem [65]. A continuous-media application that uses some Symphony class can, when it requires much CPU time, hinder other applications that also want to use the service (QoS crosstalk).

2.9 The Tiger Video File Server

Bolosky *et al.* [12] describe the Tiger distributed video file server. This server implements the storage facilities for Microsoft's Netshow video server. The goal of the project is to be able to service a large quantity of users (*e.g.* more than 10,000 users).

Tiger is constructed from a number of independent nodes or *cubs*. The collection of cubs and a Tiger video-file-server controller form the entire storage service. All cubs are connected to the outside world by an ATM network and all cubs are connected to each other by means of a switched control network. Each of the cubs hosts a number of disks.

Tiger files are distributed to all disks. The file is broken up in smaller pieces and distributed across the cubs and disks in a logical order. The idea of this type of assignment is that load is automatically balanced over all disks when data is played back: since all files are distributed to all disks, all clients need all disks for playback.

To protect against failures, data is mirrored. When a disk has failed, the contents of the failed disk is reconstructed by reading the redundant data. Tiger de-clusters mirrored data to a number of disks instead of a single predefined disk. If a disk fails, the mirrored data can be found on a large collection of disks. The advantage of this approach is that not half of a disk's bandwidth needs to be reserved for the possibility mirrored data is requested; only the *de-cluster* factor worth of bandwidth needs to be reserved per cub and disk. The mirrored data itself is kept on the inner zones of disks; the primary copies of the data are always served from the fastest zones on a disk.

Tiger implements two playback modes: single- and multibit rate mode. In the single-bit rate mode only a single bandwidth is supported by the system. If a stream uses less bandwidth than the base bandwidth, the disks are underutilized. In the multibit rate mode, the used buffer size is adjusted to reflect the bandwidth requirements. It is unclear how Tiger constructs a disk schedule for the multibit rate mode. Also, it is unclear how Tiger handles variable bit rate streams (such as Motion-JPEG and MPEG).

The research contribution of the Tiger file system is the way the distributed disk schedule for the cubs is maintained. By using the distributed disk scheduler it has become possible to use a large number of cubs in parallel to provide a single service.

Given that the multibit rate mode only exists on paper and the lack of support for variable bit rate streams makes this system less interesting. The use of a fixed schedule rather than a dynamic scheduler is identified as a problem by Sha *et al.* [92]. They found that hand-crafting real-time schedules into a slot-based schedule is usually a time-consuming and tedious task.

System	Layout	Caching/Buffering	Scheduling
UNIX FFS	BE	LRU	FCFS
UNIX LFS	MM	LRU	FCFS
UNIX EFS	MM	LRU	FCFS
VxFS	MM	LRU	unknown
Anderson [2]	CM	CM	Greedy/Cycle
Rangan [80]	CM	CM	unknown
LCMSS	CM	unknown	Cycle
CNCMS	CM, MM?	unknown	unknown
Calliope	CM	CM	Cycle
USD	MM	n.a.	preemptive EDF
Symphony	MM	MM	JIT
Tiger	CM	CM, MM?	Cycle

Table 2.1: File-system overview. The following acronyms are used: Best-Effort (BE), Least-Recently-Used (LRU), First-Come-First-Serve (FCFS), Mixed-Media support (MM), Continuous-Media support (CM), Not-Applicable (n.a.)

2.10 Summary

There are many best-effort file systems, continuous-media file systems and a few systems that can store both storage classes. Best-effort file systems are probably not suitable for storing continuous-media files unless they are radically changed. The primary problem with existing best-effort file systems is that these systems optimize for the case that only the request latency is important.

Continuous-media applications require QoS guarantees in terms of bandwidth and deadlines. If requests do not complete before some user-defined time, it is possible that the user notices a hiccup in the stream. For all continuous-media servers that are presented, the scheduling of disk requests is important to meet all real-time demands.

Given that continuous-media file systems require vast amounts of disk resources, it can be expected that such systems also store best-effort file systems. There are only a few systems that allow both storage classes on the same disks, such as the USD and Symphony. USD lacks a true nonpreemptive disk scheduler, and Symphony implements an integrated server. A symbiosis of the two systems is required.

Table 2.1 presents an overview of all reviewed file systems and their support for mixed-media file storage.

Chapter 3

File-system traffic

This thesis addresses combined (digitized audio and video) continuous-media and best-effort file-system storage. Continuous-media data requests are requests that originate from the recording or playback of digitized audio and video; best-effort data requests are requests that are generated by best-effort applications such as UNIX file systems. The distinction between the two classes of file-system traffic is in the timing. In the former data blocks need to be served at the correct pace so that playback or recording can execute smoothly without hiccups. For best-effort file systems it is important to execute the requests as quickly as possible: most likely a user is waiting for the completion of the request. However, no dramatic error occurs when it takes somewhat longer for a request to complete.

There are two types of continuous-media data streams: constant bit rate and variable bit rate streams. Constant bit rate streams are streams in which each sample or frame has the same size, which implies that resource requirements to store or retrieve such a stream are predictable. When a constant bit rate stream is compressed, redundant information is removed from the stream and since the amount of redundancy can vary over time, the resulting (compressed) stream is usually a variable bit rate stream. In the variable bit rate case, each frame or sample can have a different size and the amount of resources that are required to playback or record such a stream is not constant. However, the amount of resources that a variable bit rate stream requires is (much) less than what is required for an uncompressed constant bit rate stream.

There exist many types of best-effort traffic and it is impossible to enumerate all of them. Best-effort traffic requests are all the requests that do not have explicit timing constraints on them.

To study the nature of best-effort data traffic, best-effort systems can be traced. A system trace is a log that records all events that take place on the system. A file-system trace, for example, records all file-system operations that have taken place on the system while it is executing. The description of these operations are recorded in *trace files*. These trace files can later be used to determine bandwidth, latency, load, burst lengths and idle periods in the best-effort load.

When the trace files contain enough information, the trace files can also be used to re-run the original requests on another configuration of the system. This is important to answer so-called ‘what-if’ questions: new policies and algorithms can be tested and compared to the original system, and the performance effects of the changes can be measured relative to the original

system. This can be done in a simulator (see for example [15]) or in a real system.

There are not many file-system traces available. There are two reasons for this: (1) it is a tedious job to collect all of the system traces and (2) the traces may contain (company) sensitive information and are not to be made public. It is hard work to gather all of the trace files because of the amount of state that needs to be recorded non-obtrusively. To collect information from, for example, a disk, the disk driver needs to be altered and every request that passes through the disk driver is logged in a trace file. The amount of information that needs to be kept is usually large (especially when traces are created that need to be used for re-runs) and one needs to make sure that recording log entries does not interfere with normal processing. In the example of the disk driver, one needs to make sure that writing back the trace file to a disk does not hinder the traced disk driver.

Since all or most of the operations are recorded when trace files are generated, a complete system state is available. This can be considered sensitive because the trace file can contain hidden information such as file names, system structures or implementation details. Because of this, commercial firms usually do not export trace files that are made on their system; competitors could take advantage of this.

The downside of the unavailability of commercial traces is that only public domain traces can be used to analyze traffic and this leads to a limited view of system load. In fact, most available file and disk system traces are generated by the computer-science community itself – we know perfectly what our own load looks like [74, 6, 87].

The remainder of this chapter describes continuous-media data and best-effort file traffic in more detail.

3.1 Continuous-media traffic

Continuous-media data is a digital representation of analogue audio and video. For digital audio, a continuous-media stream consists of a series of digital audio samples, sampled at a fixed frequency (*e.g.*, 44.1 KHz for CD quality audio). Digital video is a series of digital representations of the color contents of individual frames. These frames are displayed at a fixed frequency (*e.g.*, 25–30 frames per second).

In its raw form continuous-media data is a constant bit rate stream. Each sample or frame has the same size and since the sampling frequency does not change over time, the resulting stream requires a constant throughput over time. This makes throughput requirements predictable and resource allocation straightforward.

However, when continuous-media data is not compressed, it can be quite voluminous. Audio requirements range between 8 KB/s for telephone quality audio to 187.5 KB/s for DAT quality audio. Video requires more bandwidth, ranging from 500 KB/s for a video stream of $320 \times 160 \times 8 \times 10$ (number of horizontal pixels, number of vertical pixels, bits per pixel and frames per second) to 112.5 MB/s for a $1280 \times 1024 \times 24 \times 30$ video stream. Clearly, it is quite costly in terms of required bandwidth and storage capacity to store continuous-media data in its raw form.

Continuous-media data lends itself to considerable compression before a significant loss in quality becomes apparent. When data is compressed, redundant information is removed and only

the essential information is kept. There exist numerous compression techniques and an overview of most techniques can be found in Aalmoes *et al.* [1].

When continuous-media data is compressed, it is quite likely that the resulting data stream is no longer a constant bit rate stream. If, for example, silence detection is used on an audio stream, the stream has different resource requirements for sound and silence. For video, some parts of the stream compress better than other parts, which also results in a variable bit rate stream.

Two popular video compressions schemes are Motion-JPEG [102] and MPEG [59]. Motion-JPEG compresses individual frames by removing redundancy on a per-frame basis. MPEG also removes temporal redundancy by comparing frames with earlier and later frames to detect similarities. Motion-JPEG can be used for interactive purposes since it only requires a single frame to perform a compression or decompression: the end-to-end latency between source and sink is not considerably increased by the compression or decompression technique. Since MPEG requires a number of past and future frames to compute a frame, its use for interactive sessions is limited: the end-to-end latency is considerably increased.

The Huygens laboratory at the University of Twente primarily uses Motion-JPEG hardware. A number of measurements are performed to the nature of Motion-JPEG compressed video streams to get a feel for the amount of required resources. Since MPEG uses similar compression techniques as Motion-JPEG, it is only extended with temporal compression techniques, it is assumed that from a storage perspective, requests to store or retrieve MPEG-compressed video streams are of similar nature compared to Motion-JPEG compressed video streams. The only difference is the amount of required resources.

3.1.1 Motion-JPEG

To understand the bandwidth requirements of Motion-JPEG compressed movies, a number of measurements are performed with Motion-JPEG compressed movies. In particular, the sizes of the Motion-JPEG compressed frames are measured.

The Huygens laboratory at the University of Twente uses AVAs and ATVs [78, 79] to record and render digitized audio and video. An AVA is device that is located between an ordinary (CCD) camera and an ATM link. It digitizes analogue audio (DAT and CD quality) and video (RGB or YUV in 8, 16 or 24 bits per pixel) and it is able to compress the video by using Motion-JPEG compression [102]. Once activated, an AVA produces a continuous flow of AAL5 packets with digitized audio and video on an ATM network. An ATV is the reverse device: it renders all AAL5 packets it receives through an ATM link on a television set. Note that the AVA and ATV do not use any synchronization: they simply digitize the current frame or render the current ATM input.

The AVA digitizes video on a tile-by-tile basis. The AVA samples 8 scan lines and compresses the sampled scan lines in tiles of 8×8 pixels. The compressed tiles are copied to ATM AAL5 packets. The amount of data in a single AAL5 packet is determined by a user defined *pack factor*, which represents the number of tiles that can be packed into a single AAL5 packet. When compression is disabled, the AVA copies the uncompressed scan-lines to the AAL5 packets in quantities of 8×8 pixels. The advantage of this approach is that end-to-end latency between source and sink is minimized: rather than having to wait for digitizing an entire frame, data is transmitted as soon as 8 scan lines are sampled.

To digitize audio with an AVA, an A/D converter samples the analogue audio and copies the data into AAL5 packets that are sent as independent packets across the ATM network. A typical audio fragment holds approximately 21 ms worth of audio.

The AVA is configured to compress the digitized images through Motion-JPEG and to send the compressed frames on an ATM virtual circuit to a measurement machine. A measurement application re-assembled AAL5 packets into digitized frames and records the frame sizes. Statistics are gathered by post-processing the recorded video frame sizes.

The amount of data that is generated for a stream is determined by the sample size of a frame (the number of scan-lines and the number of horizontal pixels), whether or not horizontal down-sampling is used,[1] the quality factor of the Motion-JPEG compression and the frame rate. A standard PAL signal consists of 288 scan lines and 768 horizontal pixels at a rate of 50 fields per second. These 50 fields are divided into 25 even and 25 odd fields that are *interlaced* on a display. The AVA is only capable of compressing 25 fields per second.

Three movies are used to determine the bandwidth requirements of Motion-JPEG compressed video with an AVA. The first two videos are based on an analysis of the movie the 'The Hunt for Red October' [72] and the last movie is based on 'Wallace and Grommit: A Close Shave' [75]. The 'Hunt for the Red October' is analyzed at the maximum rate: no horizontal down-sampling (*i.e.*, all X-pixels are digitized) is used and the maximum frame rate and sampling size is used. The movie runs for approximately 72 minutes and a total of 108,000 frames are analyzed. In the second try horizontal down sampling is enabled and the frame rate and sampling size are left unchanged. The total number of frames that are analyzed is 6,900, which represents 4.5 minutes of video. Lastly, 'A Close Shave' is digitized at 25 frames per second, 768×288 and horizontal down sampling is enabled. This movie consists of about 47,000 frames and the movie duration is approximately half an hour. All movies are sampled at the same quality.

Figure 3.1 presents the frames sizes for each of the three analyzed videos over time. The figure shows that when no horizontal down-sampling is used each frame has a size of 33(3.3) KB and when horizontal down sampling is used,[2] the frame size is 10(1.4)–11(2.3) KB. The first measurement is worse than is expected by the second two measurements because when this movie was recorded, there was a bad connection between the VCR and AVA, which caused lots of signal noise. It is well known that noise does not compress well. Over all, Motion-JPEG is not really a variable bit rate stream: the frame sizes remain approximately constant over time.

3.1.2 Inter-block delays

When an audio or video stream is recorded or played back on a file system, data is first buffered in memory before it is sent to disk or to a network. Since the frame sizes are reasonably constant over longer periods, it is expected that when using large buffers, the *inter-block delay*, the time between two successive disk blocks, becomes constant. To measure this inter-block delay for Motion-JPEG compressed videos, the frames are fed into a simulator that simulates the filling of memory buffers. A number of different buffer sizes are simulated.

[1] A down-sample factor of 2 means that every 2nd pixel is used.

[2] Standard deviations are presented between parentheses.

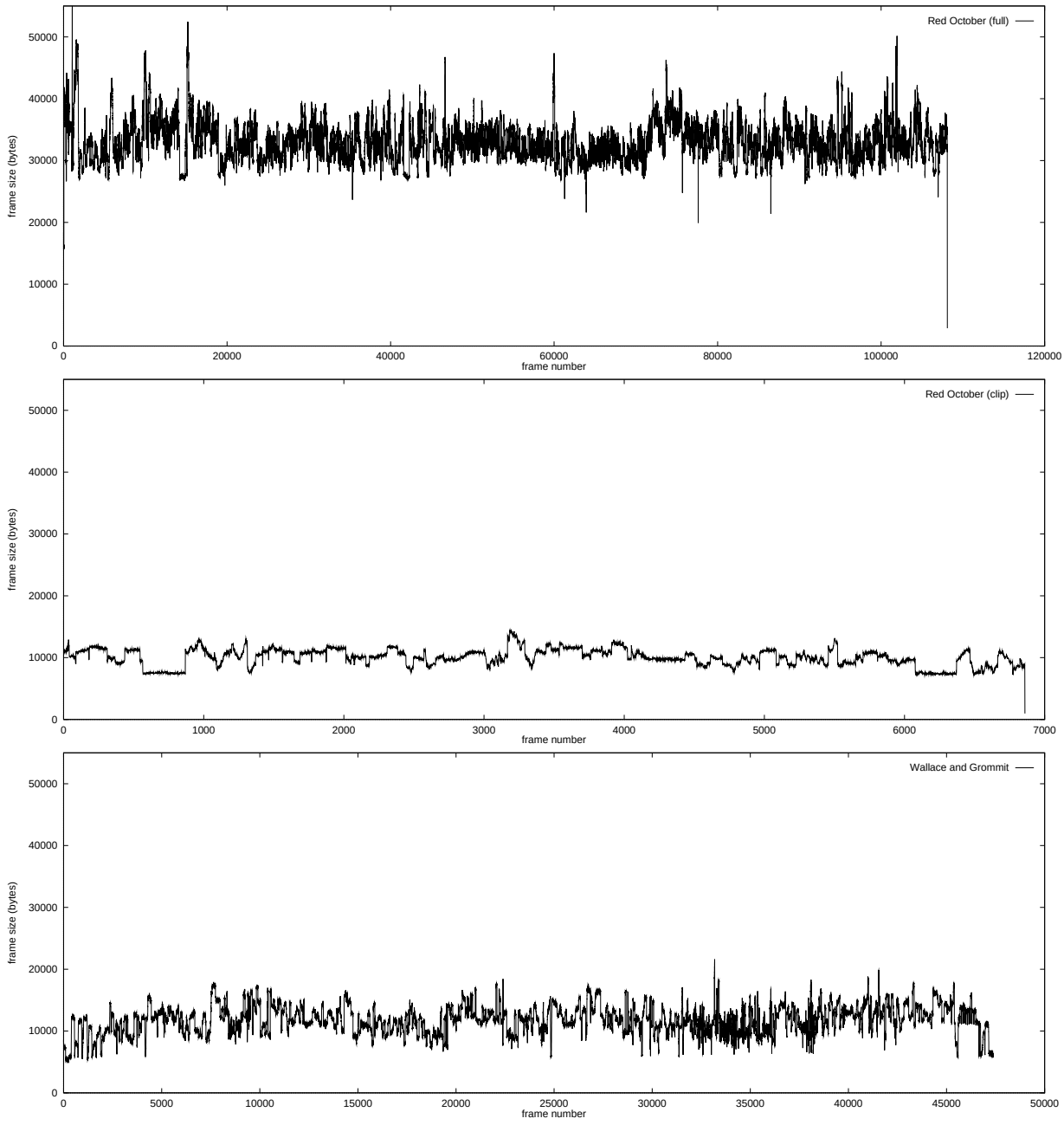

Figure 3.1: From top to bottom: the frames sizes for the entire ‘Hunt for Red October’, the frames sizes of a clip of the ‘Hunt for Red October’ and the frame sizes of the entire ‘Wallace and Grommit: A Close Shave’ movie.

Figure 3.2 presents the cumulative distribution of the inter-block delay for the three movies for a number of different block sizes, normalized to microseconds per KB buffer space. Intuitively, microseconds per KB represents the time in microseconds to fill a single kilobyte of buffer space.

The figures clearly demonstrate that larger blocks stabilize the inter-block delay for all three movies. The larger the block size, the steeper the curve, which means that the deviations to the mean are reduced. This phenomenon is also shown in Figure 3.3 where for all three movies the normalized mean plus and minus one standard deviation are shown. The figure shows that the larger the block, the smaller the standard deviation for the inter-block delay.

The reason why small block sizes give rise to jumps in the distribution as is shown in Figure 3.2 is because buffers have a size that is in the same order of magnitude as the frame sizes and because the frame rate is discrete (25 frames per second for the example streams). Hence, the jumps are at approximately 40 ms.

Although Motion-JPEG is a variable bit rate stream, using large enough block sizes reduces the variation in a stream tremendously. When only 10 % more bandwidth than the average bandwidth of a stream is allocated, only 20 % of all requests arrive quicker than expected.

3.2 Best-effort traffic

Best-effort traffic is characterized by a number of parameters: the number of disk operations per period, burstiness, idle periods, and disk latency. A burst of disk requests is defined by a number of disk requests that are executed on a disk without the disk being idle. For convenience, a burst can also have a length of a single disk operation. An idle period is defined to be the length of time between disk bursts. Disk latency is defined to be the length of time between the queue time and finish time of a disk request.

Best-effort traffic can best be studied by analyzing earlier recorded file-system and disk traces. These traces contain information of operations that are executed on a real file system or disk. Although it is true that these traces only contain a single instance of all possible workloads, if systems are traced long enough, general load tendencies can be distilled from them.

In this section two traces are described: the Sprite file-system [6] and the HP disk traces [87]. The former contains a log of file-system server operations that were executed on the Sprite distributed file system. A total of 40 Sprite clients were reading, writing, creating and deleting files and directories on the central Sprite file server. A close analysis of the Sprite traces is given in Section 3.2.1.

The HP traces contain a log of disk operations on a shared (compute) server with many disks, a log of disk operation on a client workstation and a log of disk operations that were executed at UC Berkeley. All machines ran the HP-UX 8.02 operating system, which contains measurement points to extract file-system information. The HP traces are described in Section 3.2.2.

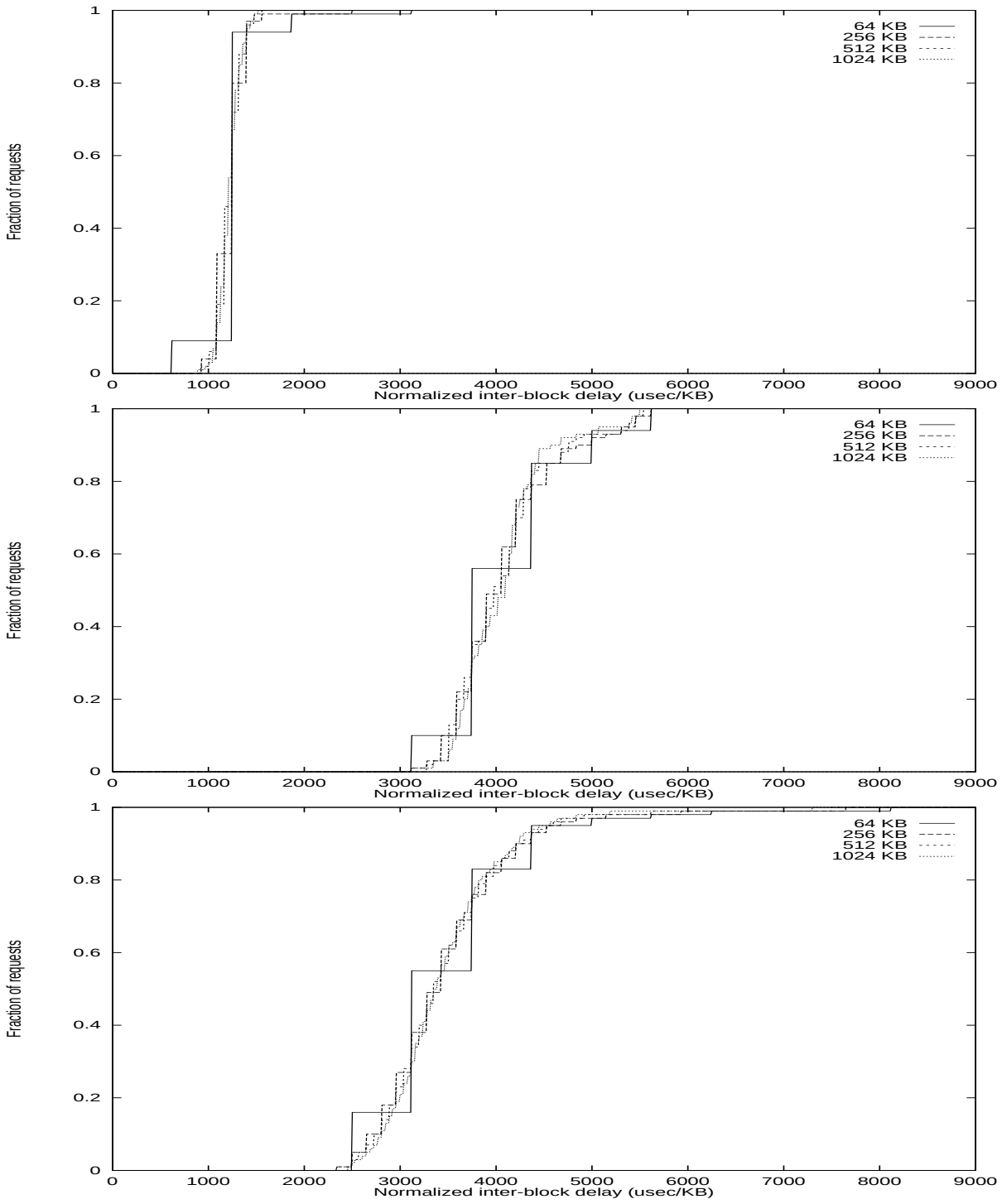

Figure 3.2: From top to bottom: the inter-block delay cumulative distribution in microseconds per KB for the ‘Hunt for Red October’ movie, a clip of the ‘Hunt for Red October’ and the ‘Wallace and Grommit: A Close Shave’ movie.

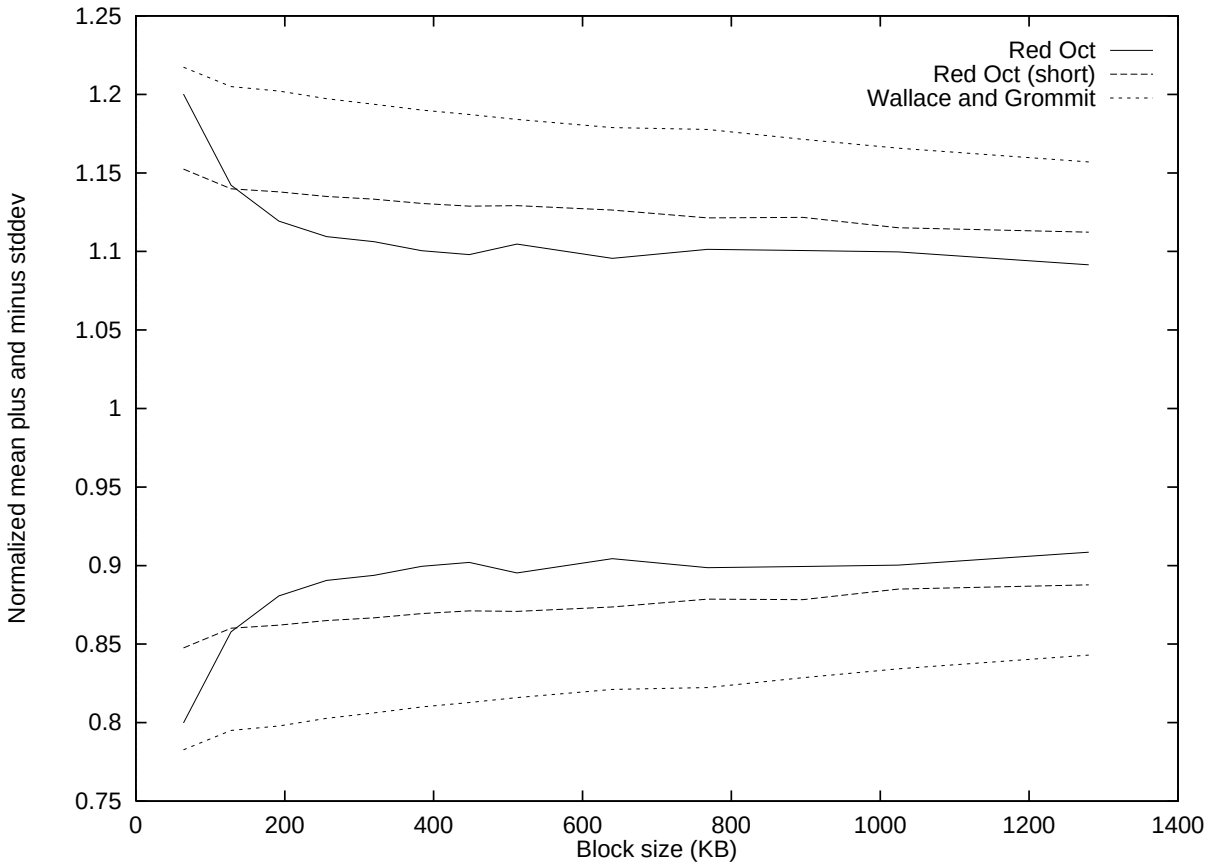

Figure 3.3: Normalized standard deviation for the three movies. ‘RedOct’ corresponds to ‘The Hunt for Red October’, ‘RedOct (short) corresponds to the short clip from ‘The Hunt for Red October’ and ‘Wallace and Grommit’ corresponds to ‘Wallace and Grommit: A Close Shave.’

	10 minutes (KB/s)	10 seconds (KB/s)
Average	8.0(36)	47.0(268)
Peak/user	458	9871
Total/all users	681	9977

Table 3.1: Sprite file-system load. Standard deviations are reported between parentheses.

3.2.1 The Sprite File-System Traces

The Sprite distributed file-system traces [6] are recorded on the Sprite distributed system. The Sprite system (client and server machines) is a system that presents itself as a single logical machine, but exists in reality of a large collection of client and server machines that run the Sprite operating system and which are connected by an Ethernet. The storage facilities in Sprite are provided by the Sprite distributed file system, which has a functionality close to that of a UNIX file system. The Sprite system consists of a few Sprite file servers (SUN 4/128) [35, 34][3] and is accessed by a group of 40 client machines. These client machines are 10-MIPS workstations and consists of SUN SparcStation 1, SUN 3, DECStation 3100 and 5000 machines.

Table 3.1 lists the average, peak and total throughput per user for two periods: a 10-minute interval and a 10-second interval. From the performance numbers can be concluded that the average load is quite low, but that there is a large deviation in the load – the load is bursty. Also it demonstrates that peaks rarely coalesce: the probability that two users generate a peak load at the same instance is low.

The number of file reads, writes and combined file reads and writes are also measured. The number of reads counted for 88 % of all operations (79 % whole file, 19 % other sequential and 3 % random), 11 % write-only (67 % whole-file, 29 % other sequential and 4 % random) and 1 % read/write traffic. Requests are considered read/write when they are read and written during a single open/close session. Whole-file access is defined by a file open, a sequential file read or write for the entire file and a file close operation. Other sequential is defined as an open followed by one or more sequential reads or writes and a file close. Interestingly enough, when the number of bytes are counted, read and write percentages are 80 % of the bytes are read (89 % whole-file, 5 % other sequential, 7 % random), 19 % of the transferred bytes are written (69 % whole-file, 19 % other sequential, 11 % random) and 1 % for read/write traffic. This means that most traffic is read-only to a whole file and that short files are read more often than longer files. On the other hand, the analysis shows that long files and in particular long data transfers has increased tremendously since the BSD file study [74] that was performed in 1985, 6 years earlier.

The client-cache size is 195 KB and 23820 KB with an average of 7110 KB. The server cache sizes are not reported, but since the machines are equipped with 128 MB of RAM, it is likely that a large portion of this memory is used for caching purposes. It is reported that although most file-system requests are read requests, many of the read requests hit in the client cache.

In summary, the Sprite study showed that traffic is quite bursty with a low average I/O rate, that most files are accessed sequentially, and that large client and server caches help in reducing the amount of traffic from client workstation to server machine (and from server to disk).

[3] Further detailed system information was obtained through personal communication with John Hartman.

CELLO			
disk	type	bus	file systems
0	HPC2474 S	SCSI-2	private NFS, swap, news binaries, root
1	HP2204	FiberLink	oldroot
2	HP2204	FiberLink	/users
3	HP2204	FiberLink	/nobackup
4	HP2204	FiberLink	/nobackup
5	HP2204	FiberLink	/usr/local/src
6	HP2204	FiberLink	news
7	HP2204	FiberLink	refdbms, NFS, swap, news, backup, swap, tmp
SNAKE			
0	Quantum PD425 S	SCSI	/, swap
1	HP97560	SCSI	/usr1
2	HP97560	SCSI	/usr2
HPLAJW			
0	HP335 H	HP-IB	/, swap
1	HP335 H	HP-IB	swap

Table 3.2: HP disk configurations.

3.2.2 The Hewlett-Packard Disk Traces

The Hewlett-Packard (HP) disk traces are logs of disk activity of production machines at the HP Laboratories in Palo Alto, California and at UC Berkeley. Three machines were traced: a server machine with 8 disks that was used by many people at HP Laboratories (CELLO), a personal workstation (HPLAJW), and a machine at UC Berkeley (SNAKE). The former two traces are available through HP Laboratories.[4] The traces were collected in April and May 1992.

The traces are primarily used as input to a disk simulator [104] and to analyze UNIX disk behavior [87]. When the traces are used for disk simulation purposes, the recorded disk traces are fed into a storage system simulator that behaves exactly like its real counterpart, *i.e.* a simulated version of the storage architectures of CELLO and HPLAJW. The measured variation between the real and simulated version of the system is at most 2 % [86].

The three machines that are used to record traces are described in Table 3.2. The machine CELLO consists of 7 HP2204 4002 RPM disks and a single HP2474S SCSI-2 4002 RPM disk with a total storage capacity of 10.4 GB. The two other configurations are used as a private workstation and for non-permanent file storage, respectively.

The first characteristic that is important is the load of the system. Table 3.3 lists for each of the three configurations the total number of I/Os that are executed during the entire trace period, the average number of I/Os per second per disk and the percentage of read requests. As is shown

[4]Contact: John Wilkes, wilkes@hpl.hp.com, http://www.hpl.hp.com/personal/John_Wilkes

Configuration	Total I/Os	I/Os per second/disk	percentage reads
CELLO	29,351,277	0.67	44 %
HPLAJW	416,262	0.04	42 %
SNAKE	12,554,885	0.77	43 %

Table 3.3: Load per disk per configuration.

in the table, the average load on a disk is not high.

Two configurations are more interesting to study because of their relatively high load: the SNAKE and CELLO trace. However, since only the CELLO and HPLAJW traces are available, only the CELLO trace is examined more closely.

Although the overall load may seem small, UNIX file-system load can be characterized by its burstiness. Ruemmler *et al.* [87] reported that approximately 80 % of the request inter-arrival times are smaller than 40 ms with peaks at 0 ms (many requests arrive at the same time) and at approximately 20 ms (consecutive requests are sent to the disk driver the moment the previous one completes).

To further investigate the burstiness of the disk arrivals, the CELLO trace is analyzed for the hourly rate and the rate per minute per disk. Figure 3.4 shows the mean and maximum I/O load in number of requests per hour for CELLO per disk during the entire 2 months' trace. The figures show that for most disks the load depends on the time of day. All disks suffer from a load peak at approximately 5:30AM. It is most likely that daily jobs, such as backing up the file systems, process the file systems for maintenance purposes. All disks also show that they are primarily busy from 8AM to 7PM. This is as expected since most people use these times as their primary work hours. Also, the load per disk can differ substantially: there are disks that rarely perform any operation while others are continuously busy. There can also be a substantial difference between the mean and maximum load throughout the period per disk, as is shown in the figures. This means that it is hard to predict how busy a system will be by just considering the average load. Finally, the maxima for the load are summarized in Table 3.4. It shows that disk 5 receives the maximum load with more than 90,000 I/Os per hour at day 44 (June 1st, 1992), from 4 to 5PM.

The load that is presented is a typical load for a file system that is used heavily by people for their daily tasks. If, however, a file system also hosts, for example, a popular Web-server, it would be loaded day and night.

When the load is examined on a minute-by-minute basis, it shows that some disks are loaded heavily and, during some periods, are kept continuously busy. Figures 3.5 and 3.6 show the mean and maximum number of I/Os that are executed on CELLO's disks between 8AM and 7PM throughout the 63 day trace period. Given that the average physical I/O time for the CELLO trace is 25.9 ms [87], an average of 2317 requests can be executed per minute.

The file system that runs on the measured machines sends requests to disk in two modes: either it sends a number of requests simultaneously to disk, or it waits for the completion of the previous disk request before sending a new request. Ruemmler *et al.* [87] found that between 10–20 % of all requests are less than 1 ms apart. These periods most likely correspond to the peaks in Figure 3.5 and Figure 3.6 and to the maxima that are listed in Table 3.5. During some

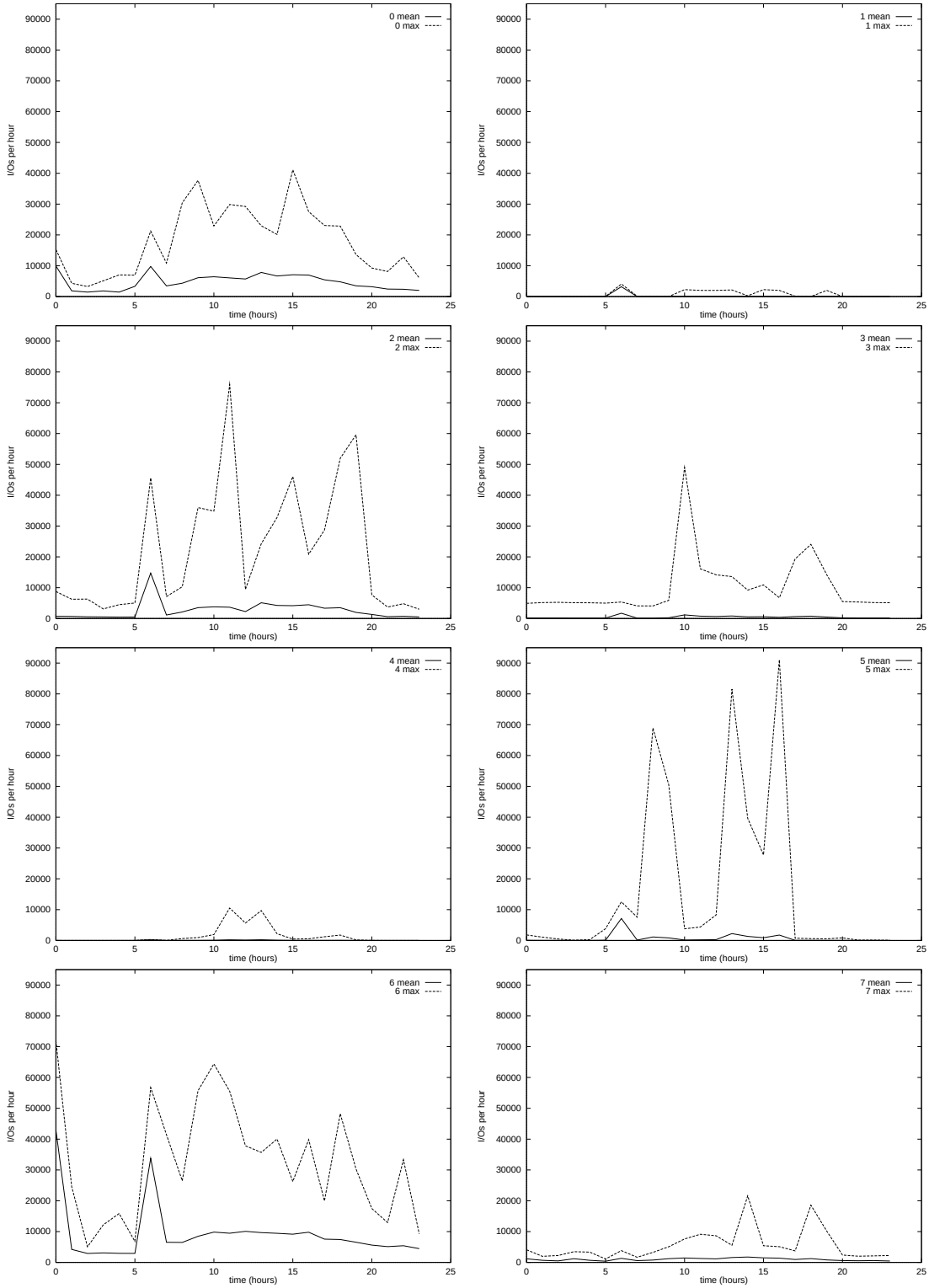

Figure 3.4: Mean and maximum load per hour on CELLO.

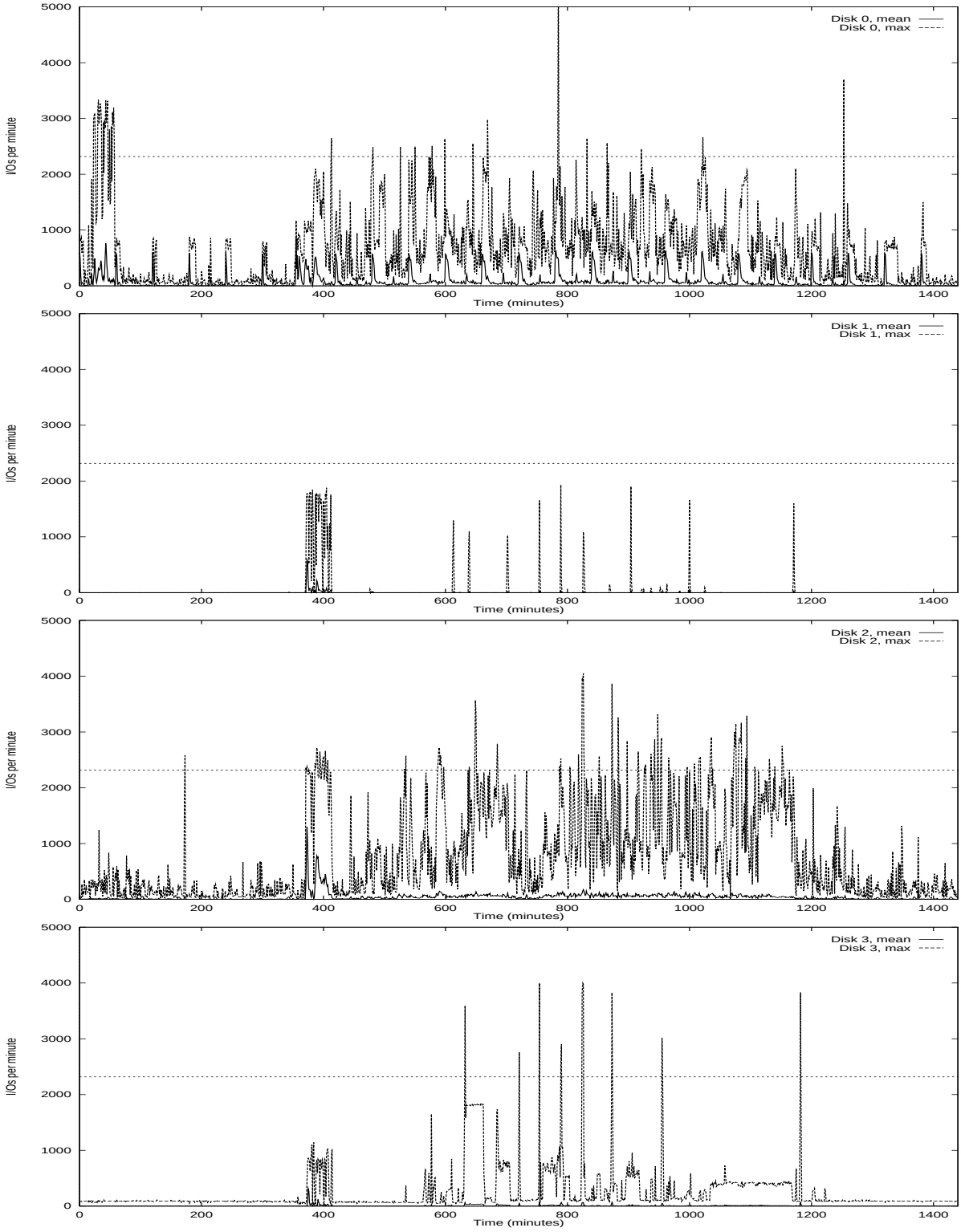

Figure 3.5: Mean and maximum load per minute for disks 0–3.

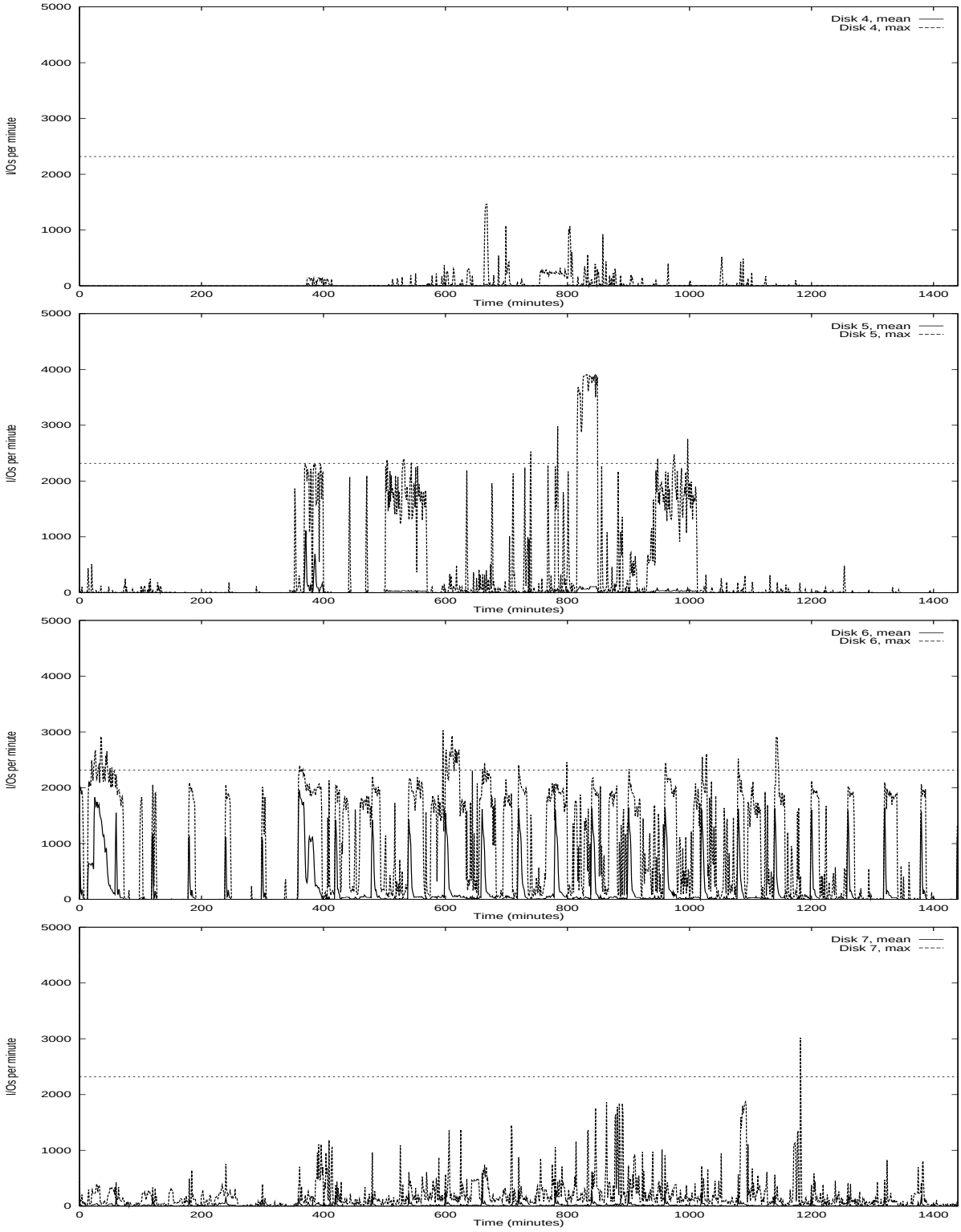

Figure 3.6: Mean and maximum load per minute for disks 4–7.

Disk	Day	Time	I/Os per hour	I/Os per second
0	44	3PM	41119	11.4
1	7	6AM	4078	1.1
2	16	11AM	75921	21.1
3	58	10AM	49025	13.6
4	3	11AM	10509	2.9
5	44	4PM	91052	25.3
6	15	1AM	72011	20.0
7	9	2PM	21588	6.0

Table 3.4: Maximum CELLO load per hour.

Disk	Day	Time	I/Os per minute	I/Os per second
0	10	13:05	5827	97.1
1	4	13:10	1956	32.6
2	55	13:46	4057	67.6
3	13	12:34	4132	68.9
4	3	11:07	1471	24.5
5	39	13:52	3908	65.1
6	47	10:12	2937	49.0
7	51	19:42	3017	50.3

Table 3.5: Maximum cello load per minute.

periods there are measured queue lengths of over 1,000 entries for the CELLO trace. However, the mean queue length for CELLO is only 8.9 (36.0) entries, the mean for SNAKE is 1.7 (3.5) entries and the mean for HPLAJW is 4.1 (7.8) entries. The second type of operations are those where requests are sent to disk once a previous request finishes. These periods do not lead to queue build-ups because at any time there is only a single request queued.

The latency of a request, the I/O request queue length and the length of bursts are interrelated. Because of the queue buildups it takes longer for an operation to finish and the latency of the request increases. Indeed, the average read latency for CELLO is 27.4 ms for read operations, and 272.0 ms for write operations (mainly caused by write bursts). Requests on CELLO take on average 164.0 ms. The average for SNAKE is 33.7 ms and 98.5 ms for HPLAJW.

Of other interest is the inter-arrival time of (groups of) requests. Ruemmler *et al.* [87] also showed, by means of the long tail from their inter-arrival distribution figure, that the inter-arrival has a large variation. This is also supported by the findings of Golding *et al.* [28] who have built *idle detectors* and *idle predictors* that can detect and predict idle periods from a disk load. It is difficult to find idle detector/predictor combinations that work well for the load.

Most of the traffic generated by the HP-UX file system is for 8 KB blocks. Since this is a relatively small block size, the physical service time is dominated by the time to seek to the desired position on disk and to wait for (on average) half a rotation. The Ousterhout BSD study [74] showed that most file-system traffic is sequential. Ruemmler *et al.* [87], on the other hand,

showed that the disk requests that are generated by the HP-UX file system are often no longer sequential anymore. On CELLO only 7.7 % of the reads and 6.3 % of the write requests are sequential accesses. This implies that, next to data layout considerations, seeks and rotational speeds are of importance.

In summary, the HP disk traces show that traffic can be quite bursty and large I/O request queues can build up from time to time.

3.3 Summary

Mixed-media workloads are predictable for the (real-time) continuous-media portion and unpredictable for the best-effort (file-system) portion. Constant bit rate audio and video streams consists of a continuous (fixed-rate) streams of digitized samples that represent the analogue audio or video. The resource requirements for such streams do not change over time.

Variable bit rate streams usually are compressed constant bit rate streams. A compressor removes redundant information from an audio or video stream in such a way that the listener or viewer does not notice the difference in quality between the uncompressed constant bit rate stream and the compressed variable bit rate stream. The compressed variable bit rate stream usually uses much less resources.

The disadvantage of compression is that the amount of resources that are required becomes much less predictable. Fortunately, when a continuous-media server uses large enough transfer buffers for, for example, Motion-JPEG compressed video streams, the variation of *inter-block* times (the time between two successive blocks) is small.

Best-effort file system traffic is of bursty and unpredictable nature. The inter-arrival time of disk requests is difficult to predict, which makes it hard to devise a scheduling algorithm that minimizes best-effort file system latencies. As is demonstrated by the Hewlett-Packard disk traces, the disk load can be quite high, with queue lengths of over 1,000 entries and a load higher than 4,000 I/Os per minute.

Since most best-effort disk traffic is for small blocks, rotational speed and the seek performance are important.

Chapter 4

I/O technology

A mixed-media file system needs to be able to record or playback a large number of continuous-media data streams simultaneously. In order to support at least 20 concurrent streams, the I/O throughput of a mixed-media file system needs to be at least 80–320 Mb/s. In this chapter, a closer look is given at how much data current hardware can move around through a machine and what the current I/O bottlenecks are. Based on this study, an efficient mixed-media file system can be designed.

Figure 4.1 shows a typical (server) workstation machine. When data is recorded, the data first arrives in the *Network Interface Card* (NIC) (*e.g.* an Ethernet or an ATM network card). This card transmits the received data across the machine's I/O interconnect to memory buffers in the main memory array and from there, the data is forwarded to the disk subsystem. Playback works similarly: data is first retrieved from disk, transmitted to memory buffers in the main memory array and finally to the client through the NIC.

The CPU does not have to be used for the actual transfer of data through the machine. As is shown by McVoy *et al.* [73] and Heybey *et al.* [101], current CPUs are not capable of transferring large quantities of data through a machine. A better approach to transfer vast amounts of data through a machine is by using the *Direct Memory Access* (DMA) functionality of interface cards to move data from and to main memory. This DMA capability allows the interface card to read and write data by itself from the main memory array.

The memory buffers that are used for the transfer can, in principle, be any memory area in the system. This means that if interface cards have on-board memory, this on-board memory can be used for I/O transfers between cards. In this case data only needs to travel once across the I/O interconnect between source and sink. The absence of on-board memory leads to a design where data travels twice across the interconnect between devices and main memory, reducing the effectiveness of an I/O interconnect roughly by 50 %.

There are many subtle device issues that can influence the overall I/O performance of a system. An example of this is when an I/O card implements the interconnect protocol poorly: all of the other I/O devices may be affected so that the I/O performance of the entire system may deteriorate. It is important to determine in detail what I/O devices and machinery are capable of, before designing a mixed-media file system.

Traditionally, the disk has always been the slowest device in a machine, which means that

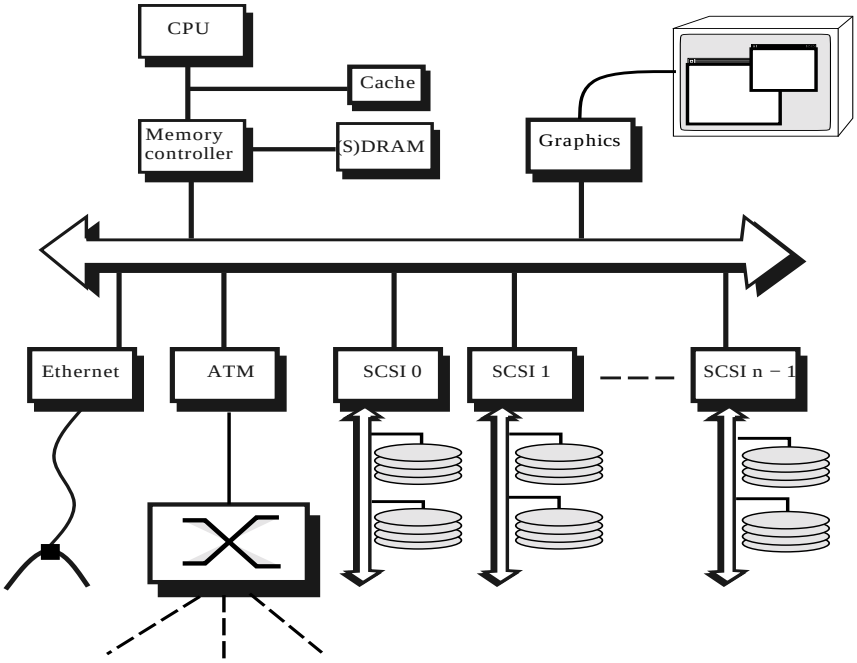

Figure 4.1: Typical hardware configuration.

the disk itself can be a bottleneck. When using a number a disks in parallel (*e.g.* in a RAID fashion [18, 76] or simply in an array without redundancy), the aggregate throughput of the disk subsystem can be increased. It is, however, important to understand *how* to connect equipment in the array: should each disk be connected by its own private interface card, or can multiple disks share an interface card?

The SCSI family of interfaces [4] is a popular disk interface. SCSI implements a (shared) bus to which all SCSI devices, including the host interface, connect. The most popular brand of SCSI today is the Fast SCSI-2 bus. This bus has an aggregate bandwidth close to high-speed disks, so if it is not carefully configured, the SCSI bus can become a bottleneck in the system.

The network and NIC are a possible bottleneck. The NIC provides an interface between the network and the host machine and if the host can transfer much more data *through* the machine rather than *in* or *out* of the machine, the network is the bottleneck. Also, it is quite common for network interface cards not to support high bandwidth transfers: a significant number of interface cards require continuous CPU attention to operate (*e.g.* most Ethernet and ATM cards require the CPU to acknowledge the receipt and transmission of each of the packets).

The I/O interconnect and memory controller that connect the NICs, the disk host interfaces and the main memory can be a bottleneck. A wide range of I/O interconnect implementations exist, but they all work on the same principle: a single bus that connects a number of I/O devices to main memory. Today the PCI I/O bus [29] is the most popular I/O bus. The PCI bus is a 32 bit wide 33 MHz I/O bus with a potential speed of 125.8 MB/s.

The last possible bottleneck is the memory itself. System memory today is managed by a memory controller that interfaces to the host bus (connecting the CPU to the memory) and

usually directly to the I/O bus. Memory speeds today range from 50–70 ns for the first word (*e.g.* 64 bits), which results in a minimum memory bandwidth of 152.5 MB/s. The second, third and fourth word can be transferred in 12–40 ns, thereby increasing the memory performance considerably for larger transfers.

It is assumed that the CPU is not in the data path of continuous-media data traffic and hence is not a bottleneck. The only task for the CPU is to orchestrate transfers between network and disk. However, in a true continuous-media file system it can be desirable to alter the stream online, such as decompressing an MPEG audio or video stream [59] for playback on client machines. In that case, CPU performance must be considered as well.

The remainder of this chapter is organized as follows. Section 4.1 describes each of I/O technologies in detail. Based on this discussion, a target machine is described in Section 4.2. To learn the actual performance of a system, real performance measurements are required. These measurements give insight which of the devices are bottlenecks, and how fast data can be moved around through a machine. Since a mixed-media file system uses the disk subsystem and network extensively, these two devices are analyzed more closely. Section 4.3 presents disk measurements and Section 4.4 presents network measurements. Section 4.5 combines the measured performance results and projects the influence of combined disk and network transfers on overall system performance.

4.1 I/O devices

This section presents a detailed analysis of a number of existing I/O technologies and a summary is given of their capabilities. In particular disks, disk interfaces, network interfaces and memory technology are described in some detail.

4.1.1 Disks

A disk is a mechanical device that consists of a number of platters on a spindle as is shown in Figure 4.2. All platters on the spindle rotate at a fixed speed (currently between 3,600 and 10,033 rotations per minute). Data is recorded on each of the platters by a read/write head and each platter is usually read and written by two heads which are located on both sides of the platter. When the platters are in motion, each head *floats* slightly above its platter. All heads together are fixed on a disk arm that can move all heads simultaneously across the disk. When a disk arm is in motion to another location on disk, the disk head is said to be *seeking*. Current seek times are between 1–20 ms and are primarily determined by the speed of the disk arm and the size of the platter. Most disks today are 3.5 inch in diameter, while new 2.5 inch disks have a potentially lower seek time.

Data on a disk is organized in sectors, heads and tracks. The minimum block size on a disk is a *sector*, which is usually formatted as a 512-byte user data block. The actual size of a sector is slightly larger to hold *Error Correcting Code* (ECC) and addressing information. A Quantum Atlas-II disk, for example, implements physical sectors of 534 bytes [43]. All sectors that are

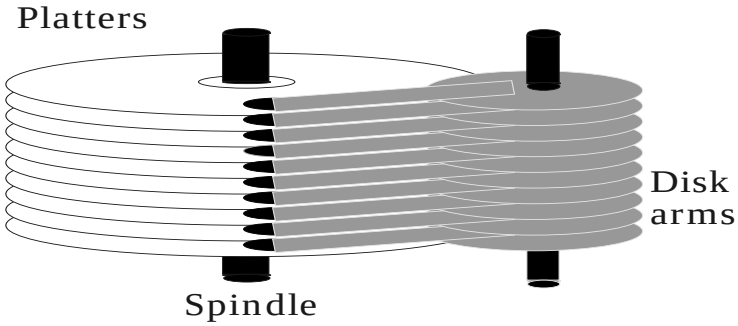

Figure 4.2: Disk mechanism.

equidistant from the spindle are called a *cylinder*. All sectors on a cylinder that share a *head* are called a *track*.

The theoretical maximum performance of a disk is determined by the rotational speed and the amount of data that is formatted on a track. A Seagate Cheetah, for example, holds 195 sectors on a track and rotates at 10,033 rotations per minute [44]. This means that during each rotation approximately 100 KB of data can be transferred. The maximum theoretical performance of this disk is 16.3 MB/s (sustained) when cylinder and head-switch times are not taken into account. It is also important to note that these speeds increase quickly (Chen *et al.* [18] claims a transfer-rate growth of 22 % per year). A Quantum Atlas-II disk [43], which is designed only a few years ago, rotates at 7,200 rotations per minute and has a maximum number of sectors per track of 179. This disk has a theoretical performance of 10.5 MB/s.

The actual performance of a disk is less than the theoretical maximum performance for several reasons. In order to find the transfer start cylinder on disk, the disk arms are moved to the desired track (*seek time*). When the disk arm arrives at the correct track, the mechanism needs to wait until the correct sector rotates under the head. This delay is called the *rotational delay* and only depends on the speed of the disk (up to 6.0 ms for a 10,033 RPM disk and 8.3 ms for a 7,200 RPM disk). Also, when a transfer of data involves two or more heads or more than a single cylinder because of the length of the transfer, other disk heads and locations need to be selected during the transfer of data (head and cylinder switches). On a Quantum Atlas-II disk the head-switch time is approximately 600 μs , a cylinder switch of a single cylinder takes approximately 1.7 ms. Note that a cylinder switch is a special kind of seek operation.

A quick analysis of a transfer of, for example, an 1 MB block of data on a Quantum Atlas-II leads to the conclusion that this operation cannot be done faster than 101.7 ms (9.8 MB/s) on the disk. When the rotational delay (on average 4.2 ms), disk interface overhead (according to Ruemmler *et al.* [86] this is 1–2 ms, NCR says that this is in the range of 2–8 ms [49]) and host-operating system scheduling are also accounted for, the Quantum Atlas-II should provide data at a rate of 9.3 MB/s (= 94.7 % of the theoretical performance).

Disk caches greatly influence the performance of disks for small I/Os. Most modern disks are equipped with 512–1024 KB of hardware disk cache with which to perform read-ahead and write-behind operations. Read-ahead means that the disk reads the next portion from the disk into its disk cache upon the completion of an earlier read request. This in the hope that the next

request from the client is a sequential one: in that case the data will have already been read and will be available without having to access the disk media. When a disk implements write-behind operations, it tries to optimize disk accesses by ordering the disk requests such that there is a minimal disk-arm movement. Such techniques can also be employed by the operating-system driver and are called *elevator-seeks* [90, 50, 81]. Note that when a disk implements write-behind techniques the disk reports completion to the host as the data arrives in the the disk cache. This, however, does not imply that the data is safe (unless the disk cache has battery backup). Disk caches do not influence the performance of large reads and writes greatly: in these cases the disk reverts to synchronous operations. Usually the operating-system's device driver cannot really control the disk caching policy; it can only switch the disk cache on or off.

Another influence on the performance of a disk is *zoning*. Zoning means that there are several zones on the disk that have a different formatting. On the outside of a disk, there is more room to format extra sectors on a track without changing the bit density much. A Quantum Atlas-II disk for example, has a maximum of 179 and a minimum of 108 sectors per track. The zone boundaries and the number of zones for disks are usually not publicly published by vendors. When such information is critical for the operation of a system, only detailed measurements can reveal the formatting of a disk. The technique of zoning is introduced recently.

Most disks recalibrate periodically to improve their internal performance. If disks are used for some time, the platters may expand or contract due to thermal effects. Drives usually know which head angle corresponds to a track number and if the platters expand, this information needs to be updated by a re-calibration. When the re-calibration is done naively, such as is the case for the Digital RZ25, the measured disk performance can vary dramatically [10]. The RZ25 recalibrates every 30 seconds even when there is work to do.

A Quantum Atlas-II disk implements a smarter policy: the disk recalibrates whenever the disk has been idle for 30 seconds. This policy works well because, as is shown by Golding *et al.* [28], when a disk has been idle for some time, it is quite likely that the disk remains idle for some longer time. This means that chances are low that new work arrives during the re-calibration. In the event that new work arrives and the disk is recalibrating, the re-calibration is terminated immediately. The re-calibration itself consists of switches between all heads to verify they are still working, internal seeks to optimize its seek profile, and checks on the disk's internal ROM.

Aggregate disk I/O performance is improved by the use of parallel disks. The *Redundant Array of Independent Disks* (RAID) technology [18, 76] uses a number of parallel disks, which can be treated as if the disks form a single large and parallel disk. Large transfers from and to disk use all disks at the same time, multiplying the aggregate bandwidth of the disks. A set of parallel blocks (*i.e.*, blocks that have the same address on all disks) are called a stripe. Redundancy is added to each stripe for two reasons. The chances of a partial disk failure of a disk array have increased compared to a single disk. Also, an array of disks has a much larger capacity compared to a single disk. Restoring the disk array from backup storage may take hours to days. The latter can lead to an enormous economic damage [104, 98]. Depending on the RAID form, a specific (RAID 0–4) or a variable disk (RAID 5–6) contains the redundant information of the data on the

other disks.[1] If any of the disks fail, the original data is reconstructed by recalculating the failed disk.

A problem with RAID technology occurs when small in-place write operations need to be performed. Rather than just updating the disk with new data, the redundant information needs to be read and written to perform the update. A small write may cause four independent I/O operations instead of a single disk update. To alleviate this problem, RAID technology is sometimes used in combination with append-only file systems (*e.g.*, a log-structured file-system). In XFS [3], for example, an entire stripe of data is collected in memory before the data is transmitted to disk. The redundant information is only computed once in memory on the entire stripe.

Another approach is by using specialized RAID hardware such as the AutoRAID [104] system from HP. AutoRAID implements two storage modes for data, based on the 'activeness' of the data. When data is not active (*i.e.*, read-only) it is stored in a RAID-5 fashion. Newly written, or re-activated data is stored on a set of disks that are organized as a mirror. The advantage of this approach is that the small write problem does not occur: such data are always written on the mirrored disks in two (parallel) I/O operations. AutoRAID provides automatic *demotion* and *promotion* of data between the two storage modes. Demotion of data from the mirror to RAID-5 only happens when (a) there is not enough disk space available to absorb traffic bursts and (b) the system is predicted to be idle. Finally, AutoRAID also provides an extra layer of addressing to hide the effect of demotion and promotion to higher layers of software (*e.g.* a file system).

4.1.2 Disk interfaces

Disk interfaces provide the interface from the disk mechanisms themselves to the host. There are several varieties of disk interfaces, but the *Small Computer System Interface* (SCSI) is currently one of the most practical ways of connecting disks to computers. SCSI implements a bus to which a number of SCSI devices can be connected and each of these devices can, in principle, communicate with each other. Devices can be magnetic and optical disks, tapes, printers, processors, CD-ROMs, scanners, medium changers and communications devices [4].

SCSI provides peer-to-peer communication between devices on the bus. Before an operation starts, a source device selects a destination device in a selection phase. Following the selection phase, messages and bulk data can be transferred between devices. SCSI devices can temporarily release the SCSI bus to allow other devices to use the bus. This option is primarily used when an operation does not produce a data stream immediately: a transfer from disk may require a disk-arm movement and a SCSI device can release the SCSI bus while it performs the seek.

Currently there are two major brands of SCSI: SCSI-1 and SCSI-2. SCSI-1 is a completely asynchronous protocol where each transferred word of 8-bits is acknowledged separately. Most SCSI-1 buses run with a 25 MHz clock and a transferred word needs to be asserted for several

[1] RAID level 0 uses a number of disks in parallel without redundancy. RAID level 1 mirrors data from one set of disks on the other disks. RAID level 2 implements a memory-style ECC on the data, which is stored on the redundant disks. RAID level 3 implements bit-interleaved parity and RAID level 4 implements block-interleaved parity. RAID level 5 is similar to RAID level 4, except that RAID level 5 rotates the redundant data over all the disks. Level 3–5 use the exclusive-or operator to calculate redundancy. RAID level 6 uses Reed-Solomon codes to protect against double failures.

clock cycles before it is acknowledged by the sink. A SCSI-1 bus performs maximally at 7 MB/s in ideal environments, but realistically it can transfer up to 4 MB/s second [47]. In SCSI-2 data can also be transferred synchronously: a number of words can be transmitted in a sequence before the first acknowledgments are received by the sender. SCSI-2 buses run at a frequency of 25 MHz (SCSI-2) or 40 MHz (Fast SCSI-2). In SCSI-2 each word (of 8 bits) needs to be asserted for at least 4 cycles. This means that a SCSI-2 bus has a maximum performance of approximately 9.5 MB/s. Wide SCSI-2 is identical to SCSI-2 except that the word size can be 16 or 32 bits, increasing the maximum performance to approximately 19.1 MB/s for plain Wide SCSI-2 and 38.1 MB/s for Fast and Wide SCSI-2.

The SCSI protocol is also implemented on coaxial and fiber links. The Fibre Channel implementation of SCSI, for example, connects up to 126 devices with an aggregate performance of 100 MB/s per link. While the bus oriented SCSI-2 suffers from short cable lengths (the longer the cable, the lower the bandwidth), Fiber Channels can be up to 30 meters when coaxial cables are used. Optical cables can even be (much) longer [96].

SCSI host interfaces are available in a wide variety for most host I/O buses. Most of these devices implement at least some part of the SCSI protocol. Some of the available SCSI cards implement an entire SCSI processor to execute the SCSI protocol [48], and some only implement the bare minimum interface thereby forcing the host CPU to run the entire SCSI protocol [47]. The disadvantage of the latter is that the SCSI performance is decreased by the overhead of interrupting the CPU for each SCSI step.

Most SCSI host cards that are available today implement DMA and *scatter-gather lists* to transfer the data from and to main memory. With DMA the CPU is only required to set up the data pointers in the card's DMA chip and to start the transfer – no memory copying is required by the CPU. Scatter-gather allows the SCSI host-cards to transfer data from and to memory in a non-contiguous manner.

4.1.3 Networks

Today many types of (high speed) computer networks exist. The network transfer rates of these networks range from 10 Mb/s for Ethernet networks to 622 Mb/s for *Asynchronous Transfer Mode* (ATM) and Gigabit Ethernet networks. The aggregate bandwidth of the network is not the only important characteristic for a mixed-media file system: what is also important are issues of what the network's policies are with respect to network traffic shaping, and how the network plugs into machines.

The Huygens laboratory at the University of Twente uses an OC-3 ATM network as the core network infrastructure with an aggregate bandwidth of 155 Mb/s. Continuous-media devices (cameras and video output devices) are used that convert the analog video signal directly into ATM cells and *vice versa*. For this reason only ATM technology is reviewed.

An ATM network consists of a number of terminals that are connected to each other by line cards in ATM switches. The switches route packets from line cards to other line cards. Each terminal is connected by a separate line card to the switch. ATM switches can be connected to each other by connecting line cards from one switch to another switch.

In ATM each *cell* is 53 bytes long, of which 48 bytes can be used for user data. The remaining 5 bytes are used to run the ATM protocol. The header contains an *end-of-packet* identifier that can be used by higher layer software and it contains information to route the ATM packet through the ATM network.

Packets are routed by a *Virtual-Path Identifier* (VPI) and *Virtual-Circuit Identifier* (VCI). An ATM signalling protocol can be used to create switched or permanent virtual circuit [19]. Switched virtual circuits are used for short duration transfers between two machines, permanent virtual circuits are used for long-term data transfers.

ATM allows QoS parameters to be assigned to virtual circuits. These QoS parameters are in terms of expected average- and peak-cell rate through ATM switches, and the expected traffic type (constant, variable, or unspecified bit rates). ATM switches use these parameters to schedule internal switch resources such as buffer capacity.

Usually the user application does not need to deal with short ATM cells. Most I/O cards have integrated support for ATM *Adaption Layer 5* (AAL5) packets. These packets can be up to 64 KBs. In AAL5, the last ATM packet is identified by the end-of-packet bit in the ATM header. Most interface cards provide segmentation into and re-assembly of AAL5 packets on ATM cells and interrupt the CPU only when an entire AAL5 packet has been transmitted or received.

4.1.4 Memory technology

The memory subsystem implements the storage and retrieval of data in the system's *Random Access Memory* (RAM) modules. Memory modules are typically organized in one or more memory banks that are selected by *row* and *column* signals. When a request for an address arrives at the memory system, the row and column signals are decoded from the address. The memory system dispatches both signals to the memory modules and 'catches' or 'inserts' data from or into the memory modules. Popular memory organizations are word sizes of 64 bits to accommodate for the 64-bit wide PC host bus and 128-bits wide memory for larger host buses.

There are basically 3 types of RAM: (asynchronous) Dynamic RAM (DRAM), Static RAM (SRAM) and Synchronous DRAM (SDRAM). DRAM is the most popular type of memory for main memory purposes: it is relatively cheap and can be made in large quantities. With DRAM it usually takes between 50–70 ns for the first word to be retrieved or stored in the memory modules. Dynamic RAM needs to be refreshed periodically because the memory is not stable by itself. DRAM is completely asynchronous; signals are not bound to any system clock.

SRAM is usually much faster than DRAM (15 ns latency is no exception), but SRAM is far more expensive. This is because it uses more transistors per memory cell and more logic is required to control SRAM. SRAM is not often used for the main memory array; it is mainly used for processor caches or video memory. SRAM is fully synchronous on the system clock [41].

SDRAM is an approach that combines both types of memory. Internally, SDRAM uses DRAM, but to external clients the memory appears to be synchronized to the system clock. Synchronizing all inputs and outputs to the system clock simplifies the design of the chip-set and memory interface, enabling it it to be based on simple state machines. Also, because control lines are not level driven, but controlled by the system clock, designers can achieve tighter specifications [40]. Most Pentium-II based machines are now equipped with SDRAM.

The oldest existing type of DRAM is *Fast Page Mode* (FPM) memory. When a memory controller receives an address, it decodes *Row Address Select* (RAS) from the address and it drives RAS to the main memory array to select the desired memory row. Next, *Column Address Select* (CAS) selects which column is used to insert or extract data from the array. When data is read from FPM memory, the memory output lines are only valid when CAS is low. A new column can be set when CAS is high again. Hence, FPM does not support pipelining functionality [42].

Extended Data Out (EDO) RAM is an extension to FPM memory where the word on the data bus remains valid even when CAS is de-asserted. With EDO RAM, the memory controller can dispatch the next column address while it is reading the word from the memory output lines. In *Bursting* EDO (BEDO) RAM it is not even necessary to dispatch the next column address, simply asserting CAS instructs the memory to select the next word [42]. Current existing FPM, EDO and BEDO RAM can burst four words for a single RAS and usually 2 or 3 host cycles are required to retrieve the next word.

The maximum memory performance is easily calculated. The time from row selection to the first word is between 50–70 ns. If bursts are not taken into account, and when memory is considered to be 64-bits wide, the memory performance ranges between 152.6 and 109.0 MB/s for the various memory types. When bursting is taken into account FPM memory delivers the next elements from the burst in 33–40 ns, EDO in 20–33 ns and BEDO in 16–20 ns [42]. This means that the maximum throughput of FPM is 160.6–204.8 MB/s, EDO has a throughput of 180.6–277.4 MB/s, and BEDO memory 234.8–311.4 MB/s. Current SDRAM delivers the first word in 50 ns and the second, third and fourth word in 12 ns. Combined with memory organized in 64 bits per word, SDRAM has a maximum throughput of 354.9 MB/s.

Clients can access the main memory array through a memory controller. This memory controller takes care of the actual decoding of a memory address into row and column addresses, and it usually controls the host and I/O bus. On PCI based PCs, for example, PCI chip sets implement PCI, memory and host bus control; on these machines the memory controller is the central machinery that takes care of all memory I/O. Therefore, the (I/O) performance of a system is influenced by the implementation of this chip set. Usually such memory controllers implement pipelining to provide high throughput as is done in the Intel 440FX memory controller for the PCI bus [24].

4.1.5 I/O bus technology

The I/O bus implements the interface between various I/O devices and main memory. Given the maximum bandwidth of the various existing I/O buses [29, 20], it seems that such buses cannot be the cause of I/O bottlenecks in a system: a PCI bus configuration, for example, can transfer data at a maximum rate of 125.8 MB/s in its minimum configuration and a Silicon Graphics HIO bus is capable of transferring data with a rate of up to 1.2 GB/s [103]. This section describes some existing I/O interconnects: Digital's Turbochannel, a bus with a maximum performance of 100 MB/s, the aforementioned PCI bus and the Desk Area Network (DAN) [8].

Turbochannel

The Turbochannel is the I/O bus that is used in previous generation Digital workstations and servers. The Turbochannel operates at a clock speed of 12.5 or 25.0 MHz and is 32 bits wide. Turbochannel option cards, such as a NIC or a disk-interface card plug into the Turbochannel and communicate with system memory through the Turbochannel.

To communicate between source and sink on the bus, the Turbochannel has defined an I/O bus transaction. Each bus transaction allows the transmission of one or more data fields. The Turbochannel implements five types of I/O transactions:

- I/O read and write of an option (*i.e.*, interface card) by the system.
- Block I/O write to an option by the system.
- DMA read from and write to the system by an option.

The first two cases, I/O read and write, are simple cases. During these transactions, one word (of maximally 32 bits) is transferred from or to a Turbochannel option card. The minimum number of cycles for a read transaction is four: a cycle to select the option, a cycle to access the desired word, a cycle to drive the word on the bus and an inter-transaction cycle. For writes, only three cycles are required: selection, storage of the word and the inter-transaction cycle. The maximum performance for reads is thus 23.8 MB/s and 31.8 MB/s for writes.

For block I/O operations, the system can send an entire block of data to an option card with a maximum of 64 words at a time. This option is similar to an ordinary I/O write, but the system does not de-assert the select signal to indicate the availability of more data. The maximum performance of this option is 91.0 MB/s when a full block of data is transmitted.

The last two options (DMA read and write) are used by an option card to send or receive data from main memory and are permutations of the other options with respect to bus signals. A DMA transfer must be at least 64 words.

The Turbochannel cannot be used to transmit data directly between option cards. For such actions, data needs to be transmitted to main memory before it can be retrieved by another option. This means that if data needs to travel between two option cards, the effective bus bandwidth is only half the maximum bus performance.

PCI bus

Figure 4.3 shows an overview of a PCI based Pentium-Pro [23]. The host CPU is connected through a cache and a host bus to an Intel 440FX memory controller that consists of a PCI *Memory Controller* (PMC) and a *Data Bus accelerator* (DBX) [24]. The PMC implements the entire PCI functionality and drives the RAS and CAS signals to the main memory array. The DBX implements write-behind functionality for data transfers from the PCI bus to the main memory array. All I/O devices connect to the same bus and can access main memory through the PMC. The figure also shows three NCR53c810 Fast SCSI-2 PCI cards that can interface to disk mechanisms.

The theoretical performance of the PCI bus is determined by bus width and clock speed. Most PCI buses today are 32 bits wide and run at 33 MHz. Once the PCI bus has been granted to a PCI

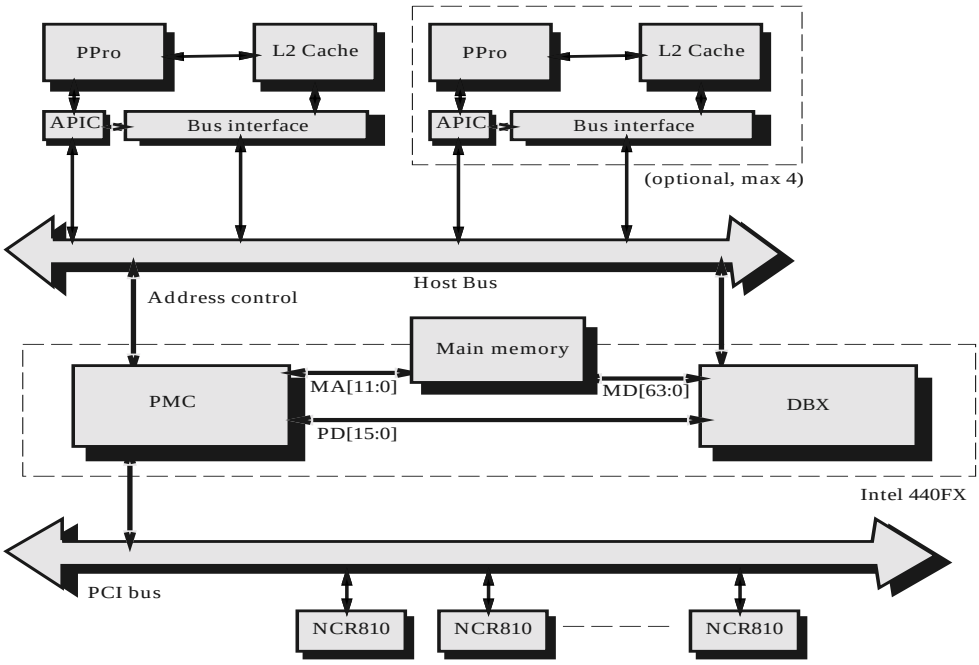

Figure 4.3: Pentium-Pro based PCI architecture.

master, a device that can send or receive data by itself, each bus cycle can be used to transfer a single word. A standard PCI bus can transfer up to 125.8 MB/s. PCI has also defined an upgrade path where the bus runs at a maximum speed of 66 MHz and can transfer 64 bits per cycle. This high-end bus has a maximum performance of 503.5 MB/s.

Before data can be transmitted across the PCI bus by a PCI bus master, the bus needs to be allocated first. For this, each PCI bus master has a private bus-request signal (REQ#) to the bus *arbiter*, to indicate its desire to use the PCI bus as a bus master.[2] The bus arbiter grants the bus to a PCI device by driving the GNT# signal.

Once the PCI bus has been granted to a master, a device asserts FRAME# to indicate its desire to read or write data from a PCI slave (target). When a bus master drives FRAME# it also drives the requested address on the data/address lines of the PCI bus. In the next bus cycle, the initiator assert IRDY# to indicate it is ready to transfer the data that corresponds to the driven address.

A PCI target responds to FRAME# by driving its DEVSEL# (device selected) signal in one (fast), two (medium) or three (slow) bus cycles. Before DEVSEL# is asserted, the target is not allowed to perform any target action. Once DEVSEL# is asserted, a target can drive TRDY# (target-ready) to indicate it is ready to perform a target action.

When both TRDY# and IRDY# are asserted, information can flow between two PCI devices. In every clock cycle the transmitting device sends a word. When either the transmitter or receiver cannot transfer the next word, it can de-assert either TRDY# or IRDY# respectively.

When the master has sent or received enough information, it can release the PCI bus by de-asserting FRAME#. This signals the bus arbiter that the bus becomes free in the next PCI cycle

[2]The symbol # shows that each device on the bus has its own line between the card and the bus arbiter.

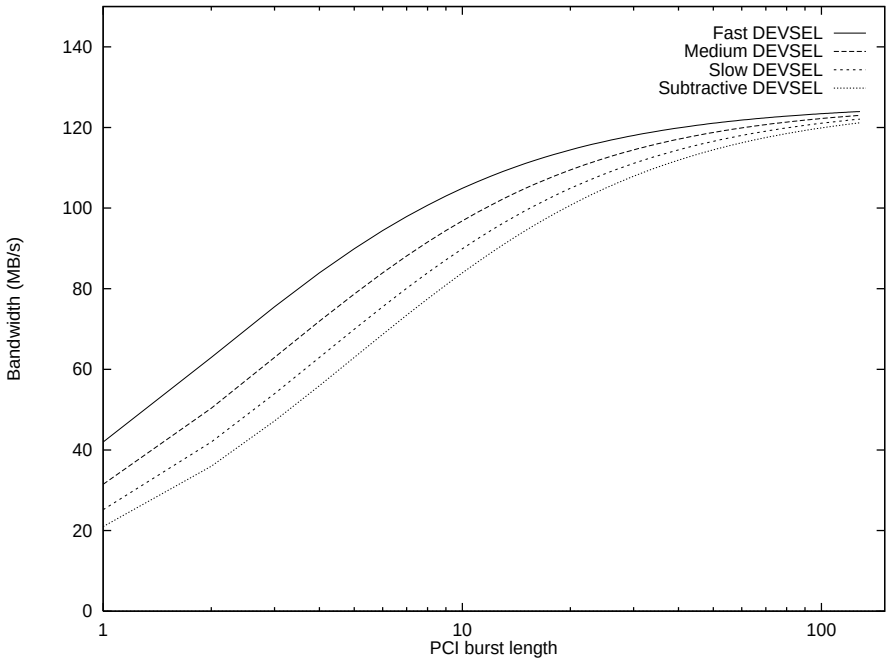

Figure 4.4: Analysis of PCI bandwidth.

and it can start a new arbitration phase – in any case, the bus is idle for at least one PCI cycle.

Before data flows between target and initiator, the initiator informs the target of its transfer intentions: *memory read* or *write*, *memory read multiple*, *memory read line* or *memory write and invalidate*. Memory read and write are intended for single word transfers. Memory read multiple is intended for bursts and memory read line is intended to transfer an entire cache-line size worth of data across the PCI bus. Memory write and invalidate informs the target (*i.e.*, the memory controller) that after writing the memory, it needs to inform the CPU caches connected to the host bus of the update. These caches can invalidate overlapping cache lines to maintain memory coherency. All but ordinary memory read and write are intended for optimized data transfer across the PCI bus.

The actual performance of the PCI bus is determined by the speed with which the device can be selected, the burst length (*i.e.*, the amount of data that is transmitted in a PCI transaction), the speed of the memory controller and the main memory array. Figure 4.4 shows the bandwidth for various values of DEVSEL# (fast of one PCI cycle, medium of two PCI cycles, SLOW of three PCI cycles and subtractive of four PCI cycles) and the burst length (a burst length of one word to 128 words), when memory controller and main memory delays are ignored. The figure clearly shows that the performance of the PCI bus quickly deteriorates if bursts shorter than 20 words (= 80 bytes) are used. More importantly: since the PCI interconnect is a bus, a single non-bursting device that is used often can reduce the throughput that is available for all devices.

Desk Area Network (DAN)

The DAN [8] is a novel approach to connect interface cards on a shared bus. The DAN is an ATM switch fabric that allows devices that are connected to the fabric to communicate directly with each other. Internally, the DAN is implemented by a bus.

The fundamental difference between the DAN and a standard bus is that DAN forces interface card designers to think more in terms of network connections than bus architectures. Instead of using the main memory array as the central connecting device, data is transferred between devices through a network connection and each network device needs enough intelligence and buffer capacity to operate on its own.

A single prototype has been implemented of this DAN and several DAN devices have been implemented by the University of Cambridge. The prototype connects the CPU cache to main memory through the switch. Also, a DAN Framestore and a DAN Digital Signal Processor (DSP) node have been implemented.

Other vendors and research groups are also working on the deployment of such network based interconnects for their machinery [36].

4.2 A measurement environment

To understand the actual issues in transferring large quantities of data efficiently through a computer, a number of performance experiments are performed on a typical PC. The goal of these experiments is threefold. The measurements give insight into the interaction between I/O devices that share an I/O bus and main memory. The results of the measurements show how to construct a mixed-media file system that is able to transfer large quantities of data. Lastly, later in this thesis performance experiments are presented with an actual mixed-media file system. Those performance numbers can be compared to the raw performance numbers that are presented in this chapter to learn of the overhead of the mixed-media file system.

The performance experiments are performed on an Intel Pentium-Pro based machine with a PCI bus for the following reasons:

- The I/O bus performance in a PC has a total throughput of 125.8 MB/s. Since most networks do not exceed a speed of 155 Mb/s, the I/O bus runs at almost an order magnitude faster than the network. Even when faster NICs become available, it is expected that such interfaces will not exceed 622 Mb/s. In that case the I/O bus will still be almost a factor of 2 faster than the network.

- PC hardware is relatively cheap. An adequate machine can be bought for less than € 1,000.

- Earlier experiments on such hardware by McVoy *et al.* [73] have shown that PC hardware can be an effective alternative to high priced and specialized workstations.

The architecture of the measurement machine is shown in Figure 4.3. This machine is equipped with a 66 MHz host bus, which is connected to the PCI bus and memory by an Intel 440FX PCI and memory controller. The machine is equipped with 128 MB of 60 ns EDO RAM.

Two storage configurations are available for the experiments. A set of 4 Quantum Atlas-II disks are available that are connected to the PC through 4 NCR53c810a Fast SCSI-2 PCI controllers [46]. Each Quantum Atlas-II disk can run at approximately the speed of a single Fast SCSI-2 bus [4], so each Quantum Atlas-II disk is hosted by a separate Fast SCSI-2 controller on the PCI bus. The second set of disks consists of 12 10,033 RPM Seagate Cheetah disks and 4 NCR53c875 Ultra-Wide SCSI PCI cards [49]. Since the performance of a single Seagate Cheetah is approximately one-third of the performance of an Ultra-Wide SCSI bus, the Seagate Cheetahs are connected to the test machine in groups of three per Ultra-Wide SCSI bus.

To measure the performance of a PC based machine, the Nemesis operating system [64] is used. Nemesis is an operating system that is especially designed to run multimedia applications by providing a kernel based deadline-dynamic scheduler. Nemesis is used for the measurements for the following reasons:

- Nemesis is small and applications can directly access the hardware when they are started with such privileges. This means that there is not much overhead when measurements are taken by user applications.
- It is reasonably straightforward in Nemesis to allocate the entire processing time to a measurement application.
- The mixed-media file system, which is presented later in this thesis, needs to run on top of Nemesis for Pegasus project reasons.

The NCR device driver that is used for the experiments is a modified public domain UNIX NCR SCSI driver. The driver is able to run the entire NCR53c8xx family of SCSI controllers. The Nemesis SCSI driver has hooks in the driver to allow encapsulation of the measurement application inside the driver – *i.e.* the measurement application is a logical part of the driver. The SCSI driver allows user applications to use the raw SCSI interface so that the user application can issue SCSI requests directly to the hardware. The control path from the completion SCSI interrupt to the measurement application is a simple one – only one Nemesis call is required to signal the completion of the task to the measurement application. This call takes approximately 45 μs .

The measurement machine is also equipped with a Digital ATMworks 350 (OPPO) ATM NIC [21]. This ATM card provides a 155 Mb/s data path to an ATM network. The NIC performs hardware segmentation and re-assembly of AAL5 frames and it can handle private data paths on a per VPI/VCI pair. The NIC is used to measure the throughput of a data stream through the network controller.

4.3 Disk I/O measurements

In theory disk performance measurements are straightforward: a measurement application simply measures the time it takes to read or write a section from disk. In practice, however, there are some performance issues that complicate the measurements. To understand what these issues are, a detailed analysis of disk operations is required.

When data is read by a measurement application from, for example, a SCSI disk, a number of actions are performed. The measurement application prepares a SCSI I/O request, which is loaded into the host SCSI controller by the operating-system's SCSI device driver. The host SCSI controller requests the SCSI bus, and when the bus is granted to the host, the SCSI request is transmitted to the destination disk. Upon reception of a SCSI request, the disk performs a number of actions: it instructs the disk heads to the requested tracks, it waits for the requested start sector to pass under the disk head (the *rotational delay*), it starts reading the data and it transfers the read data across the SCSI bus to the host. When the host SCSI controller receives the last byte, the host SCSI controller interrupts the processor to complete the transfer. The device driver handles the interrupt and wakes up the user application to inform it of the completion of the request. A write transfer works almost the same, except that data is transferred to the disk across the SCSI bus, when the disk heads are located at the desired track and sector.

Most disks today are equipped with a hardware disk cache. This disk cache is used to buffer data that is transferred from or to disk. When data is read from disk, it is first buffered in the cache before it is transmitted to the client. Alternatively, when data is written to disk, it is first saved in the disk cache before it is encoded onto the platters. The advantage of this approach is that data transfers across the SCSI bus can proceed at full SCSI-bus speed. Especially when the bus is shared by a number of disks and the disks are (much) slower than the SCSI bus, the utilization across the SCSI bus can still be high.

To learn of the actual (sustained) performance of a disk, it is important to switch off read-ahead and write-behind operations. If these are not switched off, the actual disk performance is obfuscated by these optimization techniques. For short sequential reads, for example, a measurement application would only measure the performance of the disk cache and read-ahead policy, rather than the performance of the disk itself.

As described earlier, before data is read from or written to a disk, the disk arms need to be positioned above the desired track before an operation can commence. To measure the sustained disk performance, it is important to differentiate between seek and data-transfer time. The problem, however, is that manufacturers usually do not publish detailed seek timings: only minimum, maximum and average seek times are usually available. So to be able to differentiate between seek and transfer times, detailed seek measurements are required.

To measure the seek overhead, a measurement application issues requests to perform disk arm movements on the disk and to measures how long such an operation takes. On most disks it is possible to instruct the disk heads to a certain position.

The only way to perform a seek operation on modern SCSI disks, in fact any disk operation, is by specifying the logical block address of the desired position. Internally the disk translates the logical block address to a track, head and sector number. When each track has the same number of *sectors per track* across the disk, the desired track t is calculated by $t = n/(S * H)$, where S represents the (fixed) number of sectors per track and H represents the number of heads. The requested head h is calculated by $h = (n/S) \bmod H$ and the requested sector s by $s = n \bmod S$.

When a disk implements zoning, the parameter S depends on the location on disk and logical to physical disk address translation is a non-trivial operation. Again, disk manufacturers usually do not publish the boundaries of disk zones and the number of sectors per track per zone. This implies that when a disk needs to be measured, a measurement application first needs to find

Zone	Track	NT	SPT	Capacity	Bandwidth
0	0	1772	179	1548	10.5
1	1772	352	170	292	9.9
2	2124	416	164	333	9.6
3	2540	312	161	245	9.4
4	2852	232	159	180	9.3
5	3084	416	152	308	8.9
6	3500	280	149	203	8.7
7	3780	488	139	331	8.1
8	4268	520	129	327	7.5
9	4788	208	125	126	7.3
10	4996	392	116	222	6.8
11	5388	424	107	221	6.3

Table 4.1: Quantum Atlas-II disk zones. NT represents the number of tracks per zone and SPT represents the number of sectors per track for the zone. Bandwidth represents the theoretical bandwidth of the disk in the zone.

the boundaries and number of sectors per track of the zones manually. There are several ways to find the zones on disk. A measurement application can measure how long it takes to transfer a data buffer to disk with, what is believed to be, the size of the track. If the measured time corresponds to the time it takes the disk to perform a single rotation, there is a good indication that the buffer size corresponds to the size of the track. Alternatives are to use logical block address to physical disk address translation features of disks. However, not all disks implement this translation feature.

When measuring the performance of disk read and write operations, the rotational delay also plays a role. On average, each read or write request waits half a rotation before the requested sector rotates under the head. By performing the measurements many times, the measurements must show an even distribution over the rotational delay.

For all the measurements that follow, the controller overhead and operating system scheduling overhead as described by Ruemmler *et al.* [86] are ignored. This means that all measurements include a fixed overhead.

4.3.1 Zone layouts

The Quantum Atlas-II [43] is a 4.2 GB disk with 5812 cylinders, 10 heads and 12 zones. The disk rotates at a speed of 7,200 RPM and has a maximum number of sectors per track of 179 512-byte sectors. The Quantum Atlas-II implements a SCSI call that translates a logical disk address into a cylinder, head, and sector number. By using this function, the zoning information of the Quantum Atlas-II is retrieved and is shown in Table 4.1.

In the table *zone* represents the zone number, *track* is the physical track number where the zone starts, NT represents the number of tracks of the zone, SPT represents the number of sectors per track for the zone, *capacity* represents the amount of storage in the zone in megabytes and

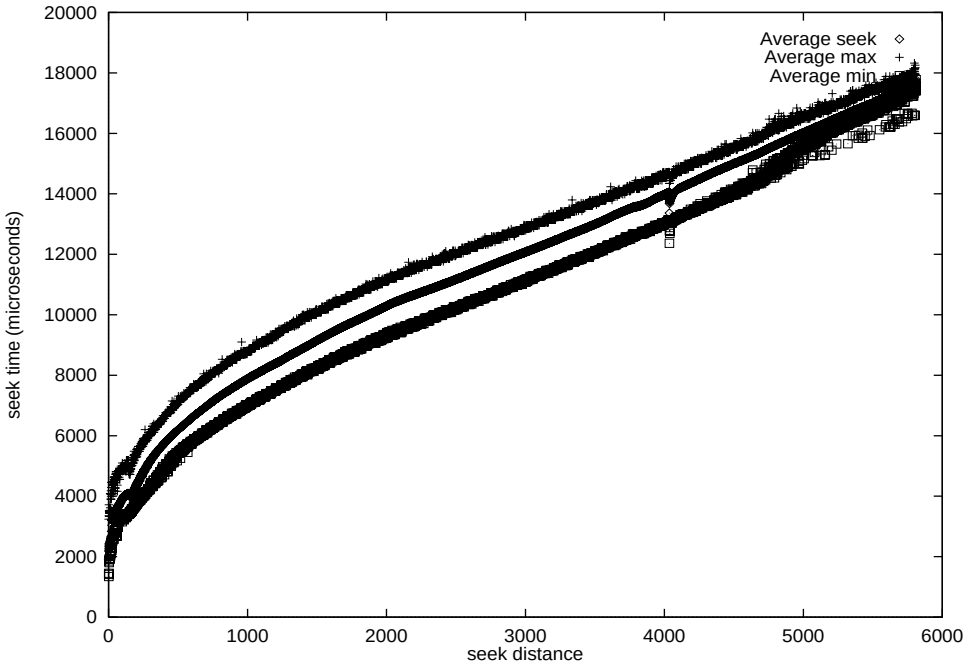

Figure 4.5: Maximum, mean and minimum seek latencies for the Quantum Atlas-II.

bandwidth represents the maximum theoretical performance for the zone in MB/s based on the rotational speed and number of sectors per track for that zone.

The outer zone on the Quantum Atlas-II disk is a relatively large portion of high bandwidth disk space. The inner regions of the disk of the disk have a lower bandwidth and the portions are relatively small.

4.3.2 Seek measurements

Based on the zone layout of the Quantum Atlas-II disk, the measurement application first measures the time of seek operations. For each track on disk, a seek to any other track larger than the begin track is performed. The assumption is made that a seek from t_a to t_b takes the same amount of time as the seek from t_b to t_a. The test is repeated several times.

Figure 4.5 shows the absolute seek times of the Quantum Atlas-II disk. The measurements show that the minimal seek (1-track), takes between 1.4 and 3.2 ms (with a mean of 1.9 ms). The published numbers for this disk specify a track-to-track seek of 900 μs . The remainder (with a mean of 1.0 ms) can be considered as controller overhead, SCSI overhead and host scheduling time.

The figure shows two phases in the seek curve: the seek over a distance smaller than 1,500 tracks and the seeks from 1,500 to 5,812 tracks. The former shows a non-linear growth of seek times that is approximated by the following formula (in milliseconds):

$$t_d = -0.0015(d - 2000)^2 + 9.5$$

Large seek timings are approximated by the formula (also in milliseconds):

$$t_d = 2d + 6.1$$

In both formulas d represents the distance in number of tracks. In any case, the minimum seek time is 1.4 ms, the maximum seek time 17.8 ms. Because of fewer measurement points at the end of the figure, the curves are much less smooth between these points.

The figure shows that there is a difference of 1.5–2.0 ms between the minimum and maximum seek time for each seek operation. These seek-time differences are most likely caused by non-deterministic behavior of the motor that moves the disk arm.

4.3.3 Single disk performance, disk cache enabled

The actual throughput of a disk is determined by several factors: how fast does the disk rotate, how many sectors are formatted per track, how much data can be sent through the SCSI bus and how fast can data be transferred between the host SCSI controller and memory.

The measurement applications measures a series of reads and writes in all zones on the disk and for a number of block sizes. The reason why zones are measured independently is to measure the actual performance in relation to the maximum theoretical performance for that zone. The reason why a number of block sizes are measured is to measure the overhead (rotational delay, SCSI bus transfers and SCSI controller overheads) of transferring data: for smaller block sizes this accounts for relatively more time than for large transfer sizes.

The first measurements are a series of reads and writes to a Quantum Atlas-II disk by the using the disk's factory settings. This means that for the initial test the read-ahead and write-behind functionality is enabled. Figure 4.6 shows the mean write bandwidth for a number of block sizes on all 12 zones on disk. It shows that only for really small block sizes, bandwidth is low in comparison to the maximum achievable bandwidth. This is because the overhead of setting up a data transfer accounts for a relatively large fraction of the total transfer time. The time to perform the storage of the data into the disk cache is the time to transfer the data across the SCSI bus (approximately 50 μs per sector on a Fast SCSI-2 bus) and time to insert the data into the cache, which is negligible. Hence, for small buffer sizes, the overhead of setting a transfer is much larger than the time to store the data.

The Quantum Atlas-II disk uses a 512 KB large disk cache, which is divided into 5 disk cache lines with a size of 89.5 KB, *i.e.* the largest track size. These disk cache lines are used to buffer data before it is written to disk. When the disk services write requests that are shorter than or equal to 5 disk cache lines worth of data for the requested zone, the write operation completes as soon as the disk has received the data in its cache. The figure shows that I/O performance almost reaches the maximum Fast SCSI-2 bandwidth.

Figure 4.6 also shows that for large buffer sizes in zones 2–11, performance deteriorates. This is caused by cluttering up the disk cache with unwritten data. In order to write new data to the disk, 'old' data needs to be flushed to disk first. The reason why zones 0 and 1 do not suffer from this effect is because the sustained disk performance roughly resembles the performance of the Fast SCSI-2 bus. In the end, most transfer speeds deteriorate to the sustained throughput of the zone.

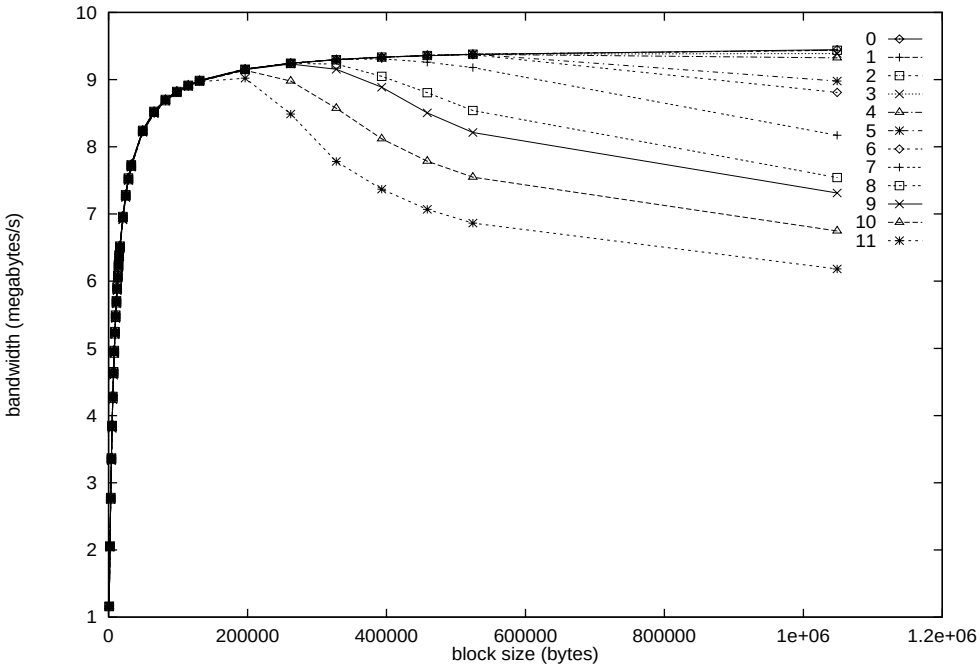

Figure 4.6: Measured write performance on a Quantum Atlas-II disk by using factory settings, *i.e.* disk cache enabled.

Since read-ahead influences are clearly undesirable for the measurements, further experiments are performed without the read-ahead algorithm. By enabling the *Force Unit Access* (FUA) bit in the SCSI read and write operations all operations complete only when data has been flushed to disk, or when data has been (re-)read from disk.

4.3.4 Single disk-read performance, disk-cache disabled

Figure 4.7 shows the I/O performance for all zones and block sizes when reading data from a Quantum Atlas-II disk without using the disk's cache. The figure shows that for small block sizes, the overhead of setting up a SCSI transfer and waiting for the rotational delay (on average half a rotation), reduces the performance of a disk tremendously. Large-block transfers perform better because the disk operates in streaming mode and the initial overhead plays a less important role in the overall performance.

According to the Quantum Atlas-II specifications, when a block of data with a size less than or equal to a full-track size is read, the disk transfers data to the host when it has read at least half of the track *and* the requested start sector has been read. This policy is called a *zero-latency read* policy.

The measured performance of the disk shows that the description is indeed correct for a transfer of less than or equal to a single track. A transfer of a data block from zone 0, for example, takes between 2.0 and 11.1 ms for a 2 sector transfer to between 15.0 and 20.9 ms for a 179 sector transfer. The SCSI overhead is approximately 2 ms per SCSI request.

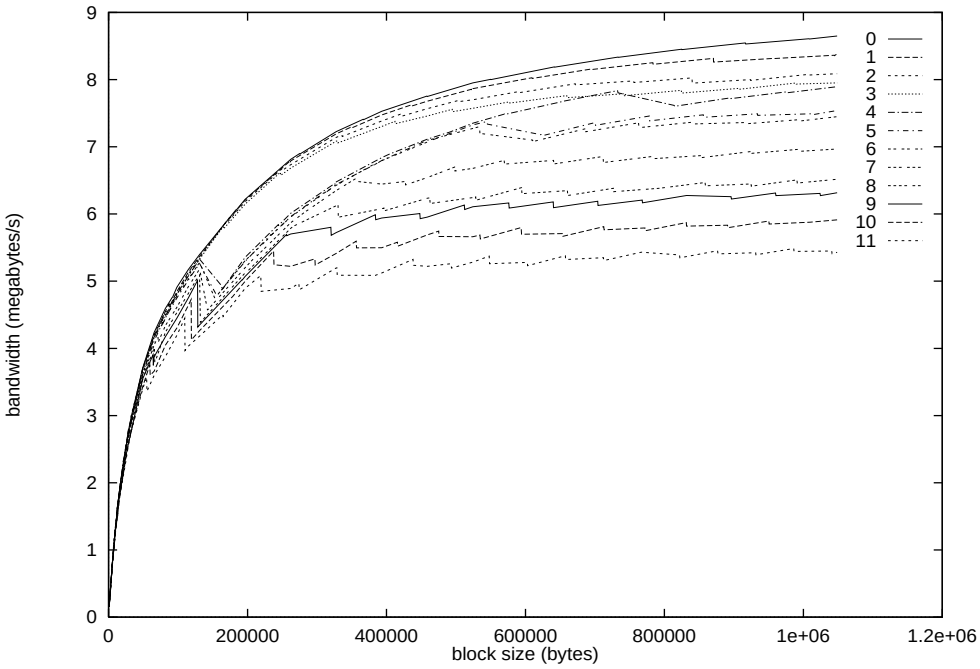

Figure 4.7: Measured synchronous read performance of a Quantum Atlas-II disk with the disk cache disabled.

Figure 4.8 shows the SCSI signal $\overline{\text{BSY}}$ (busy) for a read request of an entire track from zone 0. $\overline{\text{BSY}}$ is the or-tied signal of all SCSI bus signals. It shows that there is an initial set-up time that is used to transmit the request to the disk's SCSI controller and to have it decoded by this SCSI controller. Next, the SCSI bus is idle for 5–6 ms. In this period, the disk transfers the data into the disk cache and, when it has read half of track and the requested start sector has been found, it starts the transfer of the read data onto the SCSI bus. This is shown by the $\overline{\text{BSY}}$ signal that is active in the third and second box from the figure's right Y-axis. The SCSI transfer takes approximately 9 ms. A small period later the SCSI bus is used to transmit the final status. The total overhead consists of the time to initiate the transfer (the initial delay) and the time to complete the request (the final delay), in total approximately 2 ms.

The measured performance of a block larger than a single track in zone 0, shows that the zero-latency read policy that is used for single-track transfers is not fully used anymore. The measured performance of 180 and 358 sectors is between 15.1 and 21.0 ms and between 27.4 and 29.5 ms. A transfer of 180 sectors still uses the zero latency read policy, but a 2-track transfer does not use this pipelining technique anymore.

Figure 4.9 shows that SCSI $\overline{\text{BSY}}$ signal for a 2 track transfer in zone 0 does not become active until an entire rotation has passed in the fourth box from the left Y-axis.[3] It seems that only when an entire track is read into the cache, pipelining is used on an entire track basis: while track $n-1$ is transferred across the SCSI bus, track n is read into the disk cache. It is unclear why this disk does this: if the zero-latency read policy would not have been abandoned, the transfers would

[3] All samples show this.

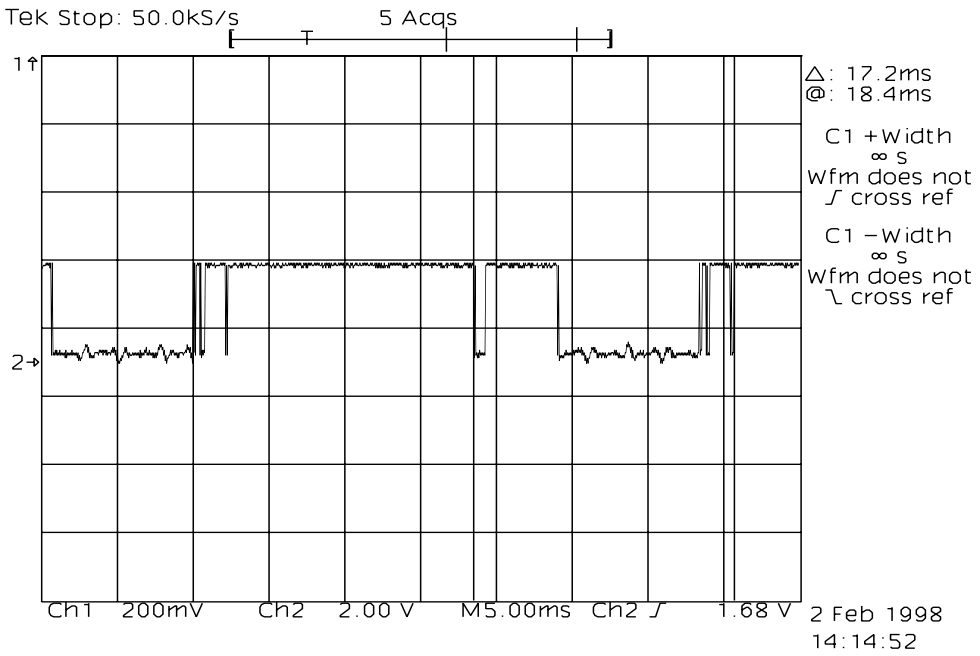

Figure 4.8: SCSI $\overline{\text{BSY}}$ signal for a single track read. Horizontal blocks represent a period of 5 ms, vertical blocks represent 200 mV. The described signal starts in the fifth horizontal block and is identified in the figure by signal 2.

have completed a few milliseconds sooner.

The inner zones show a large performance degradation between two and three track transfers. It seems that, depending on the fill rate of the disk cache, the disk reverts from a zero-latency read policy to a policy where the disk waits for the start sector of the request. It is unclear why this disk performs such a policy, except that it may be trying to make SCSI bursts as long as possible. Such a policy leads to suboptimal sequential performance in this case because the used SCSI bus is not really a fast SCSI bus; it has a performance that is comparable to that of the measured disk. It seems that this phenomenon does not occur for requests in the outer zones. In these cases, the disk cache does not drain while the disk waits for the start sector.[4]

Zones 1–3 show a slight fluctuation in the performance figure for transferring large block sizes. Here, the disk seeks to a different cylinder to complete the request. The fluctuation corresponds to a single track seek of 1.9 ms. Zone 4 (and to a lesser extent 5 and 6) shows a performance drop when an entire cylinder is transferred. It seems that at this point the SCSI bus transfers catch up with the disk and waits until the disk has enough data into the disk cache to initiate a new SCSI bus transfer. The remaining zones perform at a rate less than the SCSI transfer rate. The actual performance of the disk (including head and track switches) are reflected in the overall performance of the disk.

[4]Please bare in mind that reverse engineering internal disk algorithms by just looking at the performance numbers is difficult. A better approach is to contact the manufacturer to obtain the actual disk algorithms. However, given the responsiveness of manufacturers, it is unclear what is more difficult.

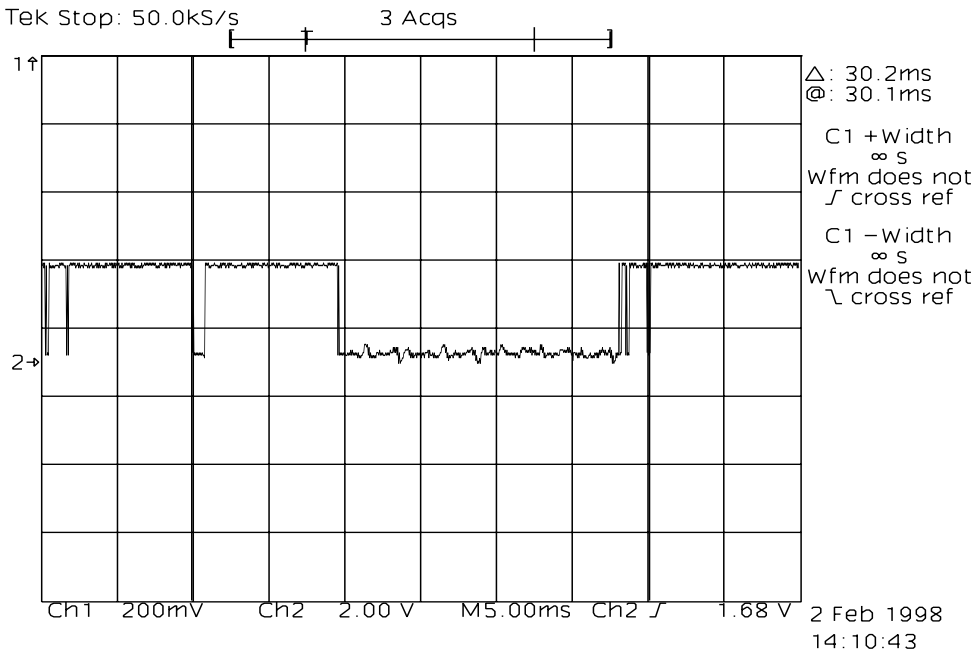

Figure 4.9: SCSI $\overline{\text{BSY}}$ signal for a two track read in zone 0. The described signal (2) starts in the third horizontal block.

4.3.5 Single disk-write performance, disk-cache disabled

Figure 4.10 shows the write performance of the Quantum Atlas-II for various block sizes in various zones. The performance for the first track (for all zones) is explained easily: as soon as the write request is decoded by the disk, data is moved across the SCSI bus to the disk cache. When the heads are at the correct track and sector, data is written from the disk cache to the disk. Zone 0, for example, shows measured values between 1.0 and 10.8 ms for a two sector write and between 10.3 and 18.7 ms for a full-track write. The maximum time for a 2-sector writes is built up from a full rotational delay (8.3 ms) and some overhead. The minimum time for a full track write is built up from some overhead plus by the time to transfer the track across the SCSI bus (9.3 ms). The maximum time for a full track write is built up from some overhead, the SCSI transfer and a full rotational delay.

When entire tracks are written to disk, the measured performance closely resembles the performance of the disk. For those zones where the disk performance is less than the SCSI bandwidth (3–11), the time to complete a request is determined by the number of required rotations, head- and cylinder switches. For zones 0–3 the request times are longer than is expected if only disk actions are accounted for. The difference is caused by rate mismatches between the SCSI and disk bandwidth: the SCSI bus is too slow to keep up with the disk. When there is no more unwritten data in the cache, the disk needs to wait (part of) a rotation before it can resume writing.

The write performance of the Quantum Atlas-II is maximized when entire tracks are written. When request sizes are not rounded up to a track size, a significant number of the requests complete in exactly the same time as the nearest larger whole-track size request. This is shown

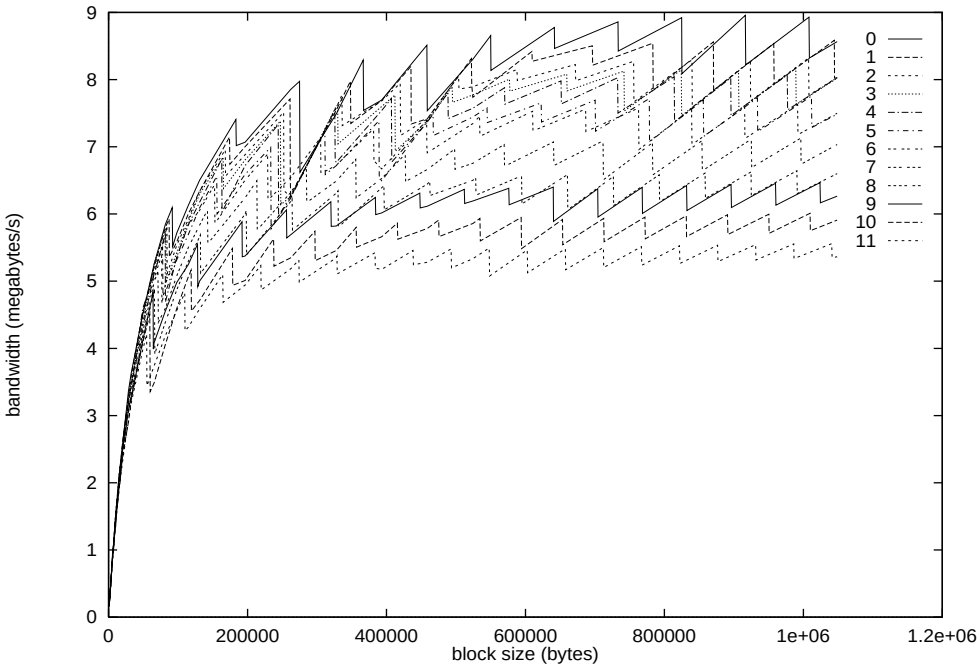

Figure 4.10: Measured synchronous write performance of a Quantum Atlas-II disk.

by the zig-zags in Figure 4.10. The size of the zig-zags differ because the number of requests that require an extra rotation is unpredictable.

Figure 4.11 and Figure 4.12 show the SCSI $\overline{\text{BSY}}$ signal for a full track write and a full track plus ten sectors write. The figures show that for writing, the disk's SCSI mechanism does not seem to burst an entire track across the SCSI bus like what is done for read transfers, or that the $\overline{\text{BSY}}$ is not well defined for SCSI write operations. Further, the figures show that for the larger request (Figure 4.12), the final completion status is transmitted an entire rotation later than the last data transfer across the SCSI bus; quite a few samples have been drawn and most samples showed such behavior.

It seems that the Quantum Atlas-II does not schedule write operations efficiently. The only conclusion that can be drawn from the samples is that the written data does not arrive in the disk cache in time to allow streaming on the disk and an extra rotation is required to update the disk. Later, in Chapter 7 it is shown that the Quantum Atlas-II's SCSI interface does not seem an efficient one: only when Ultra-Wide SCSI buses are used, the disk seems to perform reasonably.

4.3.6 Multi-disk and PCI-bus performance

A single Motion-JPEG compressed stream combined with an audio stream easily uses between 1.3 and 2.0 MB/s, which means that at most 5 streams can be recorded and played back on a single Quantum Atlas-II simultaneously. The only way to improve aggregate I/O bandwidth is by using more of these disks in a parallel manner.

Given ATM network speeds of 155 Mb/s, 2 to 3 Quantum Atlas-II disks in parallel must be

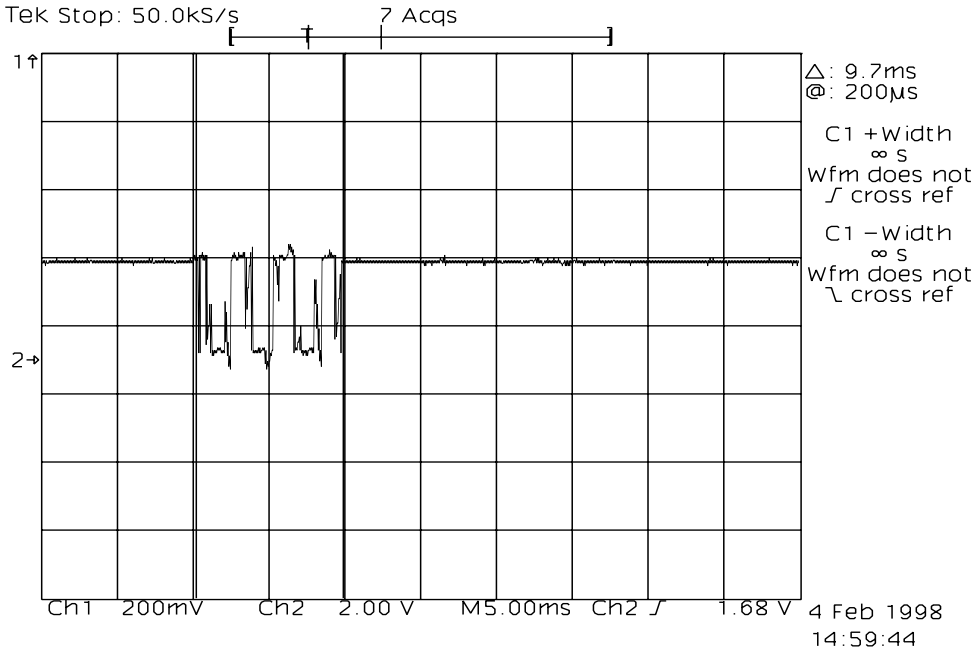

Figure 4.11: SCSI $\overline{\text{BSY}}$ signal for a single track write.

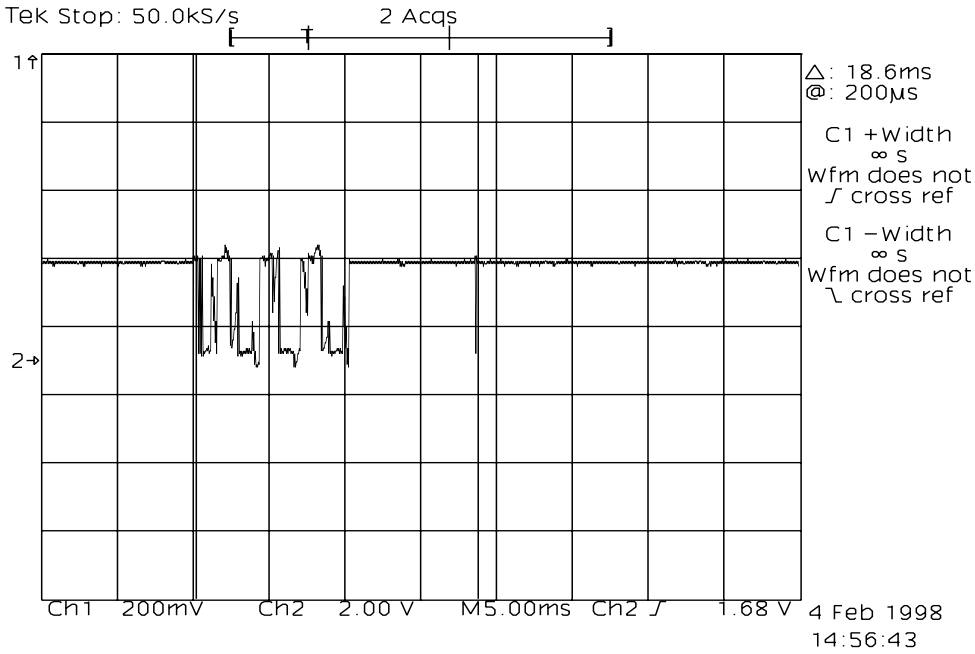

Figure 4.12: SCSI $\overline{\text{BSY}}$ signal for a single track plus ten sectors write.

able to saturate such a network. A 622 Mb/s network cannot be saturated by an I/O system that consists of NCR810a SCSI controllers and Quantum Atlas-II disks. The reason for this is that Quantum Atlas-II disks are Fast SCSI-2 disks and when these are combined with standard Fast SCSI-2 host interfaces each controller can host only a single disk. Since most PC motherboards only have 4 or 5 PCI slots, the aggregate disk bandwidth can be at most 38.0 or 47.5 MB/s.

A better combination for such a fast ATM network are the Seagate Cheetah disks that rotate with a speed of 10,033 RPM. Because of the high rotational speed and the amount of data that is formatted in the outer zone, the Cheetahs run at a sustained rate of maximally 127.4 Mb/s [44]. By combining two Cheetahs on an Ultra-Wide SCSI-2 bus, each bus can deliver at a rate of 33.0 MB/s. Given that a 622 Mb/s ATM network has a user throughput of maximally 70.4 MB/s, four Cheetahs on two Ultra-Wide SCSI buses should be able to almost saturate such a network.

Two experiments are performed. In the first set of experiments, the maximum performance of a number of parallel Quantum Atlas-II disks is measured. Later, a limited set of experiments are performed with Seagate Cheetah disks that are connected to the host through NCR53c875 SCSI cards with support for Ultra-Wide transfers. The reason for measuring the Quantum Atlas-II disks extensively despite the faster Cheetahs is because the Cheetahs became available halfway through the Quantum Atlas-II performance experiments.

To measure the maximum performance of a set of parallel Quantum Atlas-II disks, a number of Quantum Atlas-II disks are connected to the host in a parallel fashion. Each Quantum Atlas-II disk is served by a separate NCR53c810a based PCI Fast SCSI-2 controller. The measurement application and SCSI driver are modified to deal with multiple SCSI controllers simultaneously. The single disk measurements are repeated for the multi-disk configuration.

Figure 4.13 shows the speed up for 1–4 disks with varying size in zone 0 for read operations (left) and for write operations (right). The read figure shows that there is almost a linear speedup, four disks operate maximally 3.7 times as fast as a single disk. The write performance speed-up, on the other hand, demonstrates that at most two disks can be used when large blocks are written: four disks in parallel operate at 1.8 times the speed of a single disk when a megabyte block is written to all disks simultaneously. The variations for small requests are caused by an oddity in the measurement application and has no effect on large transfers.

The only shared device in the architecture is the PCI bus. Analysis on this PCI bus reveals that the SCSI PCI cards are not used efficiently. Each PCI transaction is only used for the transmission for only a few bytes.

The first performance optimization is to enable PCI burst operations to transfer a large of block of data in a single PCI transaction. The NCR53c810a PCI SCSI controller uses an on-board 64-byte deep FIFO to buffer data. When data arrives from disk into the SCSI controller, it is first stored in the FIFO before it is transmitted across the PCI bus to main memory. Likewise, when data is written to disk, it is first transferred across the PCI bus into the FIFO before it is transmitted across the SCSI bus. Long bursts are in particular important for memory read operations, which are slower than memory write operations. Memory write operations are fast because of the DBX write buffer in the Intel 440FX memory controller [24]. When longer bursts are used, the overhead of setting up a data transfer can be divided over a larger number of bytes.

Figure 4.14 shows the speed-up for a number of block sizes and a number of disks when PCI bursting on the SCSI interface cards is enabled. The left figure shows the speed-up for disk reads,

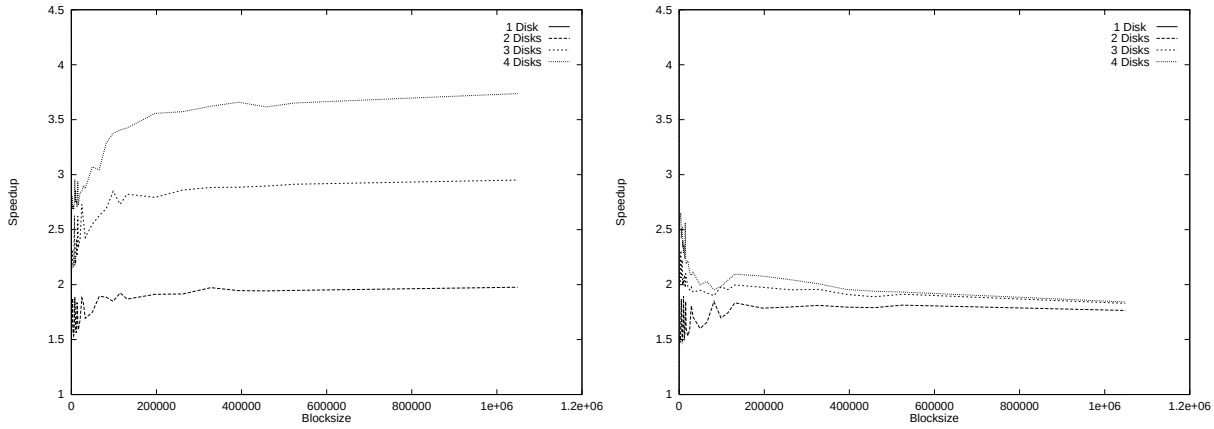

Figure 4.13: 1–4 Parallel disks for disk reads (left) and disk writes (right).

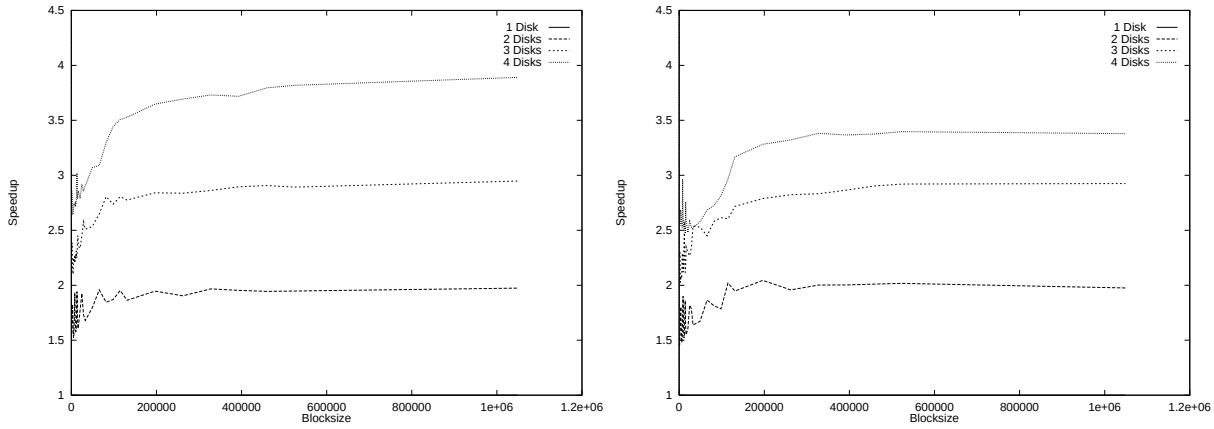

Figure 4.14: 1–4 Parallel disks, optimized PCI behavior for disk reads (left) and disk writes (right).

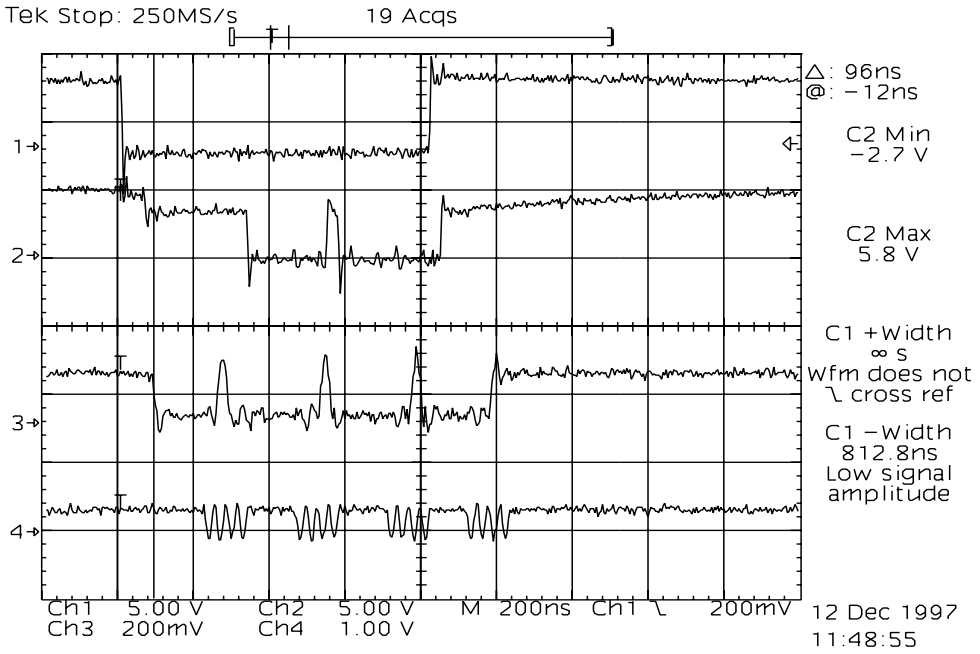

Figure 4.15: Aligned requests/memory read. Signal 1 represents $\overline{\text{FRAME\#}}$, signal 2 represents $\overline{\text{TRDY\#}}$, signal 3 represents $\overline{\text{BPRI\#}}$ and signal 4 represents RAS.

the right figure shows the speed-up for disk writes. The read speed-up shows that four disks in parallel are almost four times as fast as a single disk, and a write to four disks is almost 3.5 times as fast as a single disk. This is a tremendous performance improvement when compared to the earlier presented speed-up measurements, but there is still no linear speed up. Analysis on the PCI bus reveals that writes of 64 bytes cost 570 ns (*i.e.*, 107.1 MB/s) and reads of 64 bytes cost 1.8 μs (*i.e.*, 33.9 MB/s).

The reason for the slow memory-read operations is caused by memory alignment problems. The initial version of the measurement software only aligned the request addresses to an 8 byte boundary, which is the memory-word size. The Intel 440FX memory controller first aligns the presented memory addresses to the nearest cache-line size, which is 32 bytes on the Pentium Pro, before it starts transferring entire cache lines worth of data across the PCI bus.

The measurement application is changed so that data buffers are aligned to cache-line boundaries and the PCI signals are measured. The top two signals of Figure 4.15 show the PCI-bus signals $\overline{\text{FRAME\#}}$ and $\overline{\text{TRDY\#}}$ for cache boundary aligned memory-read data transfers over the PCI bus.[5] There are two transfers of 210 and 270 ns in which data is sent through the PCI bus as is indicated by signal 2. These periods represent the transfer of 28 bytes and 36 bytes through the PCI bus. The entire transaction takes 870 ns, which corresponds to a maximum throughput of 70.2 MB/s. The reason for two separate data transfers is because the used DRAM is not fast enough to keep up with the PCI data transfer.

[5]The PCI signal $\overline{\text{FRAME\#}}$ indicates a PCI transaction is performed, $\overline{\text{TRDY\#}}$ indicates the PCI target is transferring data.

Figure 4.15 also shows that it takes approximately 340 ns before $\overline{\text{TRDY\#}}$ is asserted on the PCI bus. The reason for this delay is that the memory controller first contacts all of the CPUs on the host bus to request the desired data from the CPU caches. If the CPU caches have a dirty copy of the requested data in their cache, the dirty data is transmitted across the PCI bus rather than the data from the main memory array. This feature enables Intel architectures to implement memory coherency without an explicit instruction to flush the CPU write buffer. If the caches do not own a copy of the requested data, they return an error indicating this [23].[6]

Lastly, Figure 4.15 shows that the NCR53c810's FIFOs are too small. The memory controller is already busy to fetch the third and fourth batch of words from memory as is indicated by the fourth signal, when the NCR controller terminates the PCI transaction. When NCR SCSI cards are used with larger FIFOs, the PCI bus is used more efficiently.

Pentium-II based machines use SDRAM instead of the Pentium Pro's ordinary EDO DRAM. SDRAM delivers the first word in 50 ns and the remainder of the memory words in a burst of 12 ns per word. A run of the measurements software on a Pentium-II based machine shows that the memory of this machine is fast enough to keep up with the PCI burst, and the short stop that is shown in Figure 4.15 is not needed. SDRAM based machines transfer 64 bytes in 840 ns, which corresponds to a transfer rate of 72.3 MB/s on the PCI bus.

The disk speedup experiment is not repeated because while analyzing the measurement results, the fourth Quantum Atlas-II disk is confiscated for ordinary storage purposes.

4.3.7 Large PCI bursts

To obtain a high I/O performance, the length of PCI bursts needs to be large. So far, the length of the PCI transfer is limited by the depth of the FIFOs on the NCR53c810a PCI SCSI-2 boards. The NCR53c810a only allows the transfer of two cache lines worth of data from and to memory.

The NCR53c875 Ultra-Wide PCI/SCSI controllers use an internal FIFO with a length of 512 bytes [49]. Given the measurements that are presented in the previous section, these controllers must be able to transfer 512 bytes in 4240 ns in combination with 50 nanosecond SDRAM. In other words, the maximum performance on a 33 MHz PCI bus should be about 115.1 MB/s for disk write operations (*i.e.*, memory-read operations). When these cards are combined with ordinary 60 ns EDO RAM, the maximum disk write performance should be slightly less: 114.6 MB/s.

A number of measurements are executed on the same measurement machine with 1–4 NCR-53c875 PCI SCSI cards. Each SCSI bus is equipped with 1–3 Seagate Cheetah disks, each capable of a theoretical throughput of 15.9 MB/s. The aggregate theoretical performance of the disks is approximately 190.8 MB/s, the aggregate performance of the SCSI buses is 152.6 MB/s and the maximum performance of the PCI bus remains 125.8 MB/s.

The first experiment is to measure the behavior of the NCR53c875 on the PCI bus. Figure 4.16 shows the PCI burst from main memory to the NCR53c875 FIFO. The figure shows that the NCR chips transfer 512 bytes per PCI transaction. An entire read transaction of 512 bytes completes

[6] The third signal in Figure 4.15 represents the host-bus signal $\overline{\text{BPRI\#}}$ (Priority-agent Bus Request), which, when activate, indicates the memory controller wants access to the host bus. The fourth signal is the main memory RAS signal.

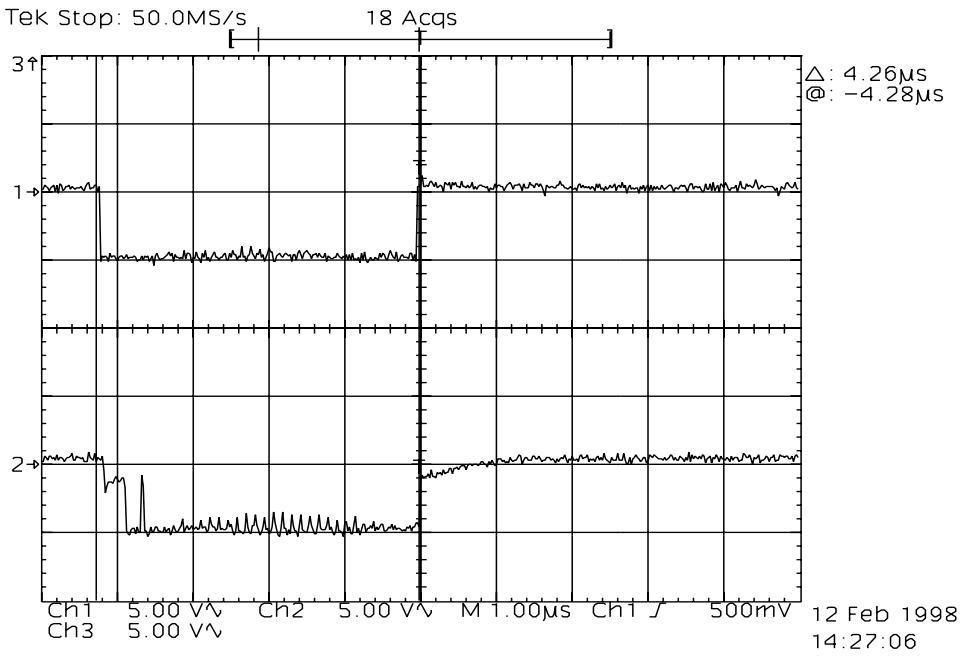

Figure 4.16: Memory read burst/NCR53c875. Signal 1 represents $\overline{\text{FRAME\#}}$ and signal 2 represents $\overline{\text{TRDY\#}}$.

in 4.26 μs , which corresponds to a maximum PCI bandwidth of 114.6 MB/s. Another experiment measures that the memory-write transfer of 512 bytes to memory completes in 3.92 μs , which corresponds to a PCI bandwidth of 124.2 MB/s. Disk-write efficiency is 91.0 % of the available PCI bandwidth and disk-read efficiency is 98.7 %.

The next experiment is to measure the aggregate performance of all disks on all 38.1 MB/s Ultra-Wide SCSI buses. Figure 4.17 shows the performance of the disks when megabyte blocks are transferred from and to disk. The horizontal bar shows the number of parallel disks, the vertical bar shows the aggregate I/O performance. The figure shows that the average measured performance of a single Cheetah is approximately 13.2 MB/s and that the maximum performance when 12 disks are used in parallel is approximately 100 MB/s for reading and 88 MB/s for writing.

Figure 4.17 shows the performance of two experiments: the experiments labelled 'Read-3' and 'Write-3' and the experiments labelled 'Read-2+1' and 'Write-2+1.' In the 'Read-3' and 'Write-3' experiment, disks are assigned naively: disk d is assigned to SCSI bus $\lfloor d/3 \rfloor$. In this case, a SCSI bus is overloaded when 3 disks are used. In the '2+1' experiment, disk d is assigned to SCSI bus $\lfloor d/2 \rfloor, d \leq 8$. Disks $d = 9 \ldots 12$ are assigned to SCSI bus $d - 8$. Hence, in this assignment, when up to eight disks are used in parallel, none of the SCSI buses is overloaded.

A single disk on a bus completes the transfer of one megabyte in approximately 75 ms (reading and writing). When a second disk is attached to the same bus, the operations complete approximately 2.5–4.0 ms later for reading and writing, respectively. This extra latency is caused by SCSI bus sharing: both disks transfer chunks of approximately 100 KB across the SCSI bus

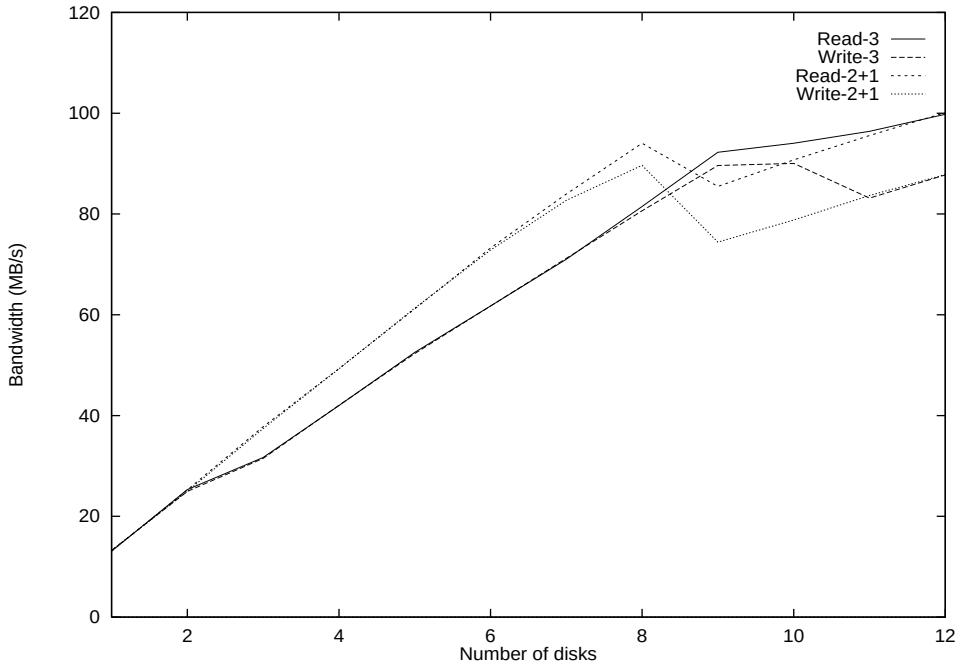

Figure 4.17: I/O throughput of 1–12 Cheetahs.

and disks wait for each other to release the bus.

SCSI buses are overloaded when three, six, nine or twelve disks are used in the ‘Read-3’ or ‘Write-3’ configuration, or when more than eight disks are used in the ‘Read-2+1’ or ‘Write-2+1’ configuration. In that case the disk mechanism is forced to wait for data or for a data drain to the host, thereby possibly missing rotations on disk. Adding the third disk on the same SCSI bus greatly reduce the performance as is shown by the ‘Read-3’ and ‘Write-3’ experiments. A megabyte transfer takes approximately 15.0 ms longer to complete. It is clearly undesirable to use three Cheetahs on the same SCSI controller at the same time.

The read time does not change much when 4–9 disks are used in the ‘Read-3’ and ‘Write-3’ experiments. When data is read from nine disks at the same time, the completion time differs only 2.9 ms from the transfer time of three disks on a single bus.

At approximately 72 MB/s, read and write performance start to differ. At this point the memory system saturates for disk-write operations (memory-read operations). Given the measured PCI timings that are presented in the previous section, it is expected that the memory saturation point is at 114.6 MB/s. The fact that the saturation point is at lower data rates may be caused by concurrent memory accesses from, for example, other I/O controllers that simultaneously use the main memory array simultaneously.

When more than nine disks are used in the ‘Read-3’ or ‘Write-3’ experiment, the total performance does not grow linearly anymore. The reason for this is that the PCI bus is fully saturated with transfers of data between the memory system and the I/O cards. The PCI effects can be simulated by a small program that prepares a PCI bus schedule for the PCI cards. Each NCR53c875 SCSI card transfers data in chunks of 512 bytes across the PCI bus. Each SCSI string can be

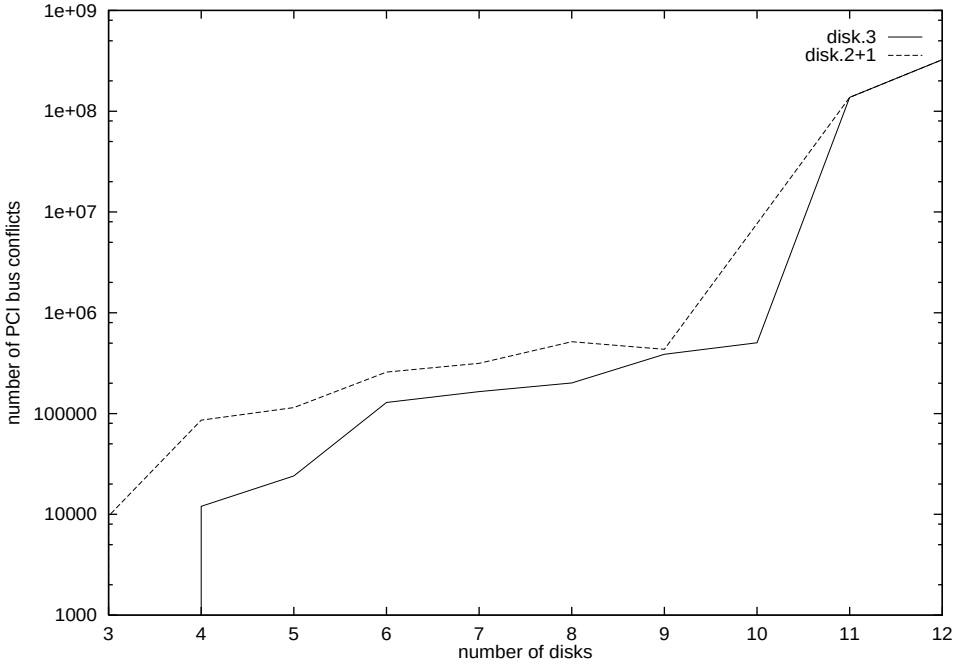

Figure 4.18: Number of PCI bus conflicts against used number of parallel disks.

equipped with one, two or three Seagate Cheetahs so each bus can deliver data at a rate of approximately 13, 24, and 32 MB/s. This means that the SCSI cards access the bus every 37.3, 19.3, and 15.4 μs and use the bus for maximally 4.3 μs for reading 512 bytes from the main memory array.

Figure 4.18 shows the number of PCI bus conflicts for each of the configurations. A conflict is defined when two or more SCSI cards want to access the PCI bus simultaneously. When there is a conflict, only one of the cards is granted the PCI bus, and the other(s) need to wait for idle time on the bus. Since the bus time of each of the SCSI cards is 4.3 μs , the others are delayed for at least this amount of time. It is unclear what the disk mechanism does when it is not served data quickly enough when writing data to disk. However, the disk needs to wait for data and it is likely that it waits for a full rotation. The disk effects have are taken into consideration for the simulation.

The figure shows that the number of conflicts for the ‘disk-3’ experiment between nine and ten parallel disks on the PCI bus almost doubles and grows exponentially between ten and eleven disks. This means that when these amount of parallel disks are used, the PCI data transfers are delayed considerably and the average performance per disk is reduced. This corresponds to the write performance drop for eleven disks as is shown in Figure 4.17.

The other experiments, labeled ‘Read.2+1’ and ‘Write.2+1’ in Figure 4.17 try not to overload the SCSI bus. Rather than using three disks on the same bus, this experiment only uses two disks on the same bus for the first eight disks. For nine to twelve disks the 3rd disks on the buses are used. The figure shows that this results in a performance gain of 6–11 MB/s when compared to the ‘Read-3’ and ‘Write-3’ experiment for the first eight disks.

It is unclear why there is a such a performance drop when nine disks are used in parallel in the '2+1' configuration. The simulation does not show an increased number of PCI bus conflicts compared to eight disks; in fact, there are even less conflicts. It is likely that the simulation is too simplistic for the complicated PCI bus and disk behavior in this particular case.[7] Beyond nine disks, the number of PCI conflicts indicates the performance loss: there are more conflicts on the PCI bus for ten disks compared to the 'Read-3' and 'Write-3' experiments and the measured performance for 10 parallel disks in the '2+1' configuration is indeed lower.

Optimizing the bus behavior means that schedules need to be sought that fit 'well' together, *i.e.* the periodicity of each of the strings need to be adjusted to each other. Unfortunately, the periodicity is either 15.4, 19.3, or 37.3 μs for this situation. Even when the periodicity can be adjusted, the start time of each of the periods is determined by a number of factors that cannot be controlled easily by the user: the disk's rotational position determines the SCSI and PCI bus-start time, the disk's SCSI chip determines when the data transfer is actually started on the SCSI bus and finally, the PCI memory controller implements a simple FCFS scheduling policy that cannot be altered by software.

Overall, throughput demanding applications can read data at a maximum rate of 100 MB/s and write data at approximately 90 MB/s, but such applications need to be aware of the reduced utilization per disk. The highest aggregate rate without really saturating the PCI bus is by using eight Cheetahs on four buses with at most two disks per SCSI string. This means that at most 94 MB/s and 89 MB/s can be transferred for disk reads and disk writes respectively. In practice, it seems wise not to use more than six Seagate Cheetahs on three Ultra-Wide SCSI controllers.

From the measured performance numbers can be concluded that PCI performance is maximized by enlarging the PCI transaction. More of the available PCI bandwidth is used to transfer data between devices. However, it is important not to overload the PCI bus. The SCSI controller's FIFOs may over- or under-run and as a result, the disks are used inefficiently. It is important to note that *all* disks perform less efficiently in that case.

To maximize the performance on a SCSI bus it is important not to overrun the SCSI bus. When the SCSI bus is overrun by multiple disks on the same bus, all disks are used less efficiently, thereby reducing the effectiveness of parallel disks.

4.4 Network performance

The continuous-media part of a mixed-media file system is, in essence, a device that copies continuous-media data from a network onto a disk and *vice versa*. The Huygens group at the University of Twente, employs OC-3 ATM networks that are capable of transferring data at a maximum rate of 155 Mb/s across the wire. The host interfaces that are used are Digital ATMworks 350 ATM PCI adapters (a.k.a. 'OPPO') for PCs and 750 ATM Turbochannel adapters (a.k.a. 'OTTO') for Digital Alpha equipment [21].

OPPO networks cards are 'smart' ATM cards: they contain enough logic to perform on-board

[7] Further measurements are required to find the causes.

segmentation and re-assembly of ATM cells into AAL5 network packets. They also contain DMA capabilities to transfer data directly across the PCI bus into the main memory array. The Nemesis driver for these network cards is written such that each data stream (VPI/VCI pair) is received on a private hardware-ring buffer with private user-data buffers. This has the advantage that under- or overflows on one ring do not hinder the transfer of data on another ring and that data is directly transferred into user memory without requiring an extra layer of indirection.

Originally, the Digital ATM OTTO card is implemented for the 25 MHz Turbochannel bus. This means that all internal timing on the network device matches the 25 MHz Turbochannel I/O bus. The OPPO card is exactly the same interface card as the OTTO with an attached 21050 PCI-to-PCI bridge chip [22] that maps the Turbochannel timings and bus accesses to accesses on the PCI bus.

To measure the performance of the ATM network, the aforementioned Pentium-Pro measurement machine is equipped with an OPPO card. Since the line rate of the OPPO cards is 155 Mb/s and since each ATM cell contains 53 bytes, of which 48 are user bytes, it is expected that at most 17.5 MB/s can be transferred between machines.

A user measurement application is constructed that sent out a large amount of data onto the ATM network. The application measures how much data can be transmitted for a number of different packet sizes. Of particular interest is the behavior of the transfer of data across the PCI bus. As is shown by the disk performance measurements, the PCI bus plays an important role when transferring data between interface card and the main memory array. If the network card does implement an efficient PCI protocol, it reduces the usefulness of the architecture.

No real effort is put into measuring the receive capabilities of the OPPO. To measure the performance of the receive side of the OPPO, it is important to know what is sent to the OPPO. This can only be done effectively by an ATM load generator, which was not available when the experiments were performed. Other machines can also be used as a traffic generator, but in that case the measurement also depends on whether or not the sender can generate a *good* load.

Although receive behavior is important to know, the measurement is not critical. It is expected that the mixed-media file system is mostly used for playback (hence, network transmit).

Also, an experiment is performed to measure the behavior of the system when more than a single virtual circuit is used. Rather than sending to a single virtual circuit, the sender sends data on a number of virtual circuits at the same time.

Figure 4.19 shows the network bandwidth for transmission of data on a number of virtual circuits and for varying network-packet sizes. The figure shows that the maximum (user) performance for larger packets is 135.7 Mb/s. In reality this means a line rate of 149.9 Mb/s. When smaller packets are transmitted, the OPPOs do not perform well and performance quickly deteriorates. The figure also shows that when more virtual circuits are used, the aggregate bandwidth of the ATM network increases.

To understand why the OPPO behaves as is shown in the figure, one needs to understand some of the internals of the OPPO card and OPPO device driver. The OPPO is used so that each virtual circuit has its own set of transmit ring buffers. When an application transmits a packet, the device driver copies the address and size into the hardware ring buffer and the device initiates a DMA to private on-board OPPO memory. Once the DMA is complete, the OPPO interrupts the processor to inform the device driver of the copy. The device sends the packet on the wire and interrupts the

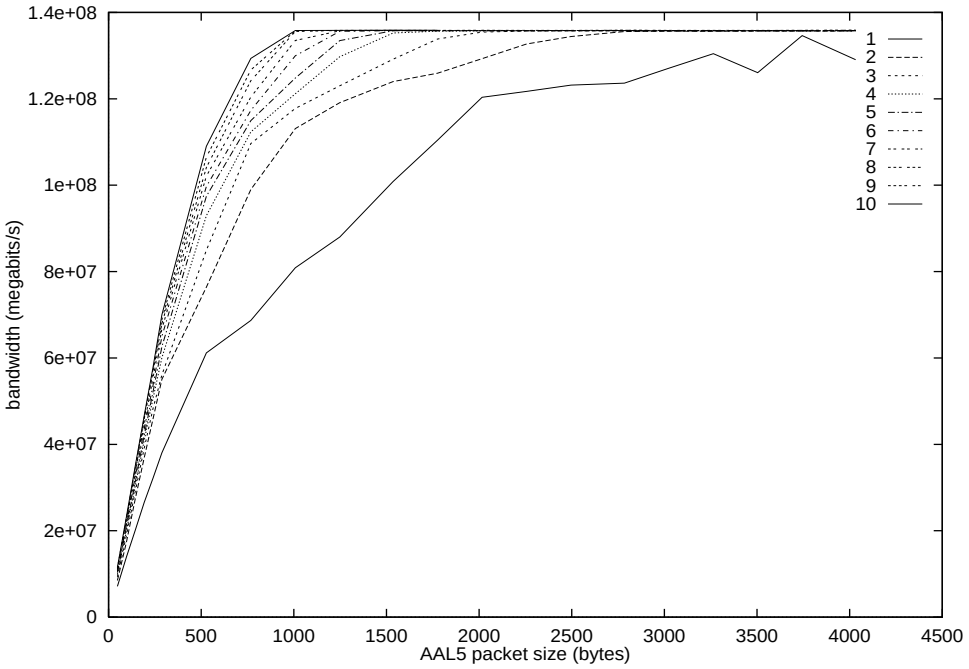

Figure 4.19: OPPO transmission performance for $1 \ldots 10$ parallel VCIs.

processor again when the transmission is complete. At this point the ring buffer becomes free again.

The reason why small AAL5 packets do not perform well is because it takes quite a while before both interrupts are dispatched to the CPU [10], the interrupt routine runs and the user application is signaled of the transmission. Current CPUs cannot handle an interrupt every 2.6 μs for 48 bytes large AAL5 packets. The measured performance for 48 byte packets is that every second approximately 18,000 packets can be transmitted, which means that the CPU is interrupted on average every 55.6 μs .

The slow packet transfers are also caused by a hardware ring buffer that is only two entries long. The OPPO DMAs all data that needs to be transmitted to a 90 KBs on-board buffer. When too much data is queued, the OPPO tries to load too much packets into its memory and simply crashes. Since the Nemesis driver does not account for the amount of queued data, limiting the ring buffer is an effective way to guard against memory overloads.

When a larger ring is used, or when more ring buffers are used by using multiple parallel VCIs, the efficiency quickly increases. The number of used hardware ring buffers is a multiplication of the number of parallel VCIs and the number of hardware-ring buffers per VCI. In the experiments more of these ring buffers are used simultaneously, and the transmission hardware is kept busy for longer periods. The hardware does not stall while waiting for the CPU to load new packet descriptor in the hardware-ring buffers.

It is unclear why only 149.9 Mb/s worth of data can be transmitted. A careful analysis on the ATM link itself is required to learn why the card cannot fully saturate the network.

As is shown by the disk performance measurements, the PCI behavior of a device is also im-

portant. If too much bandwidth is wasted when transferring data from memory to controller, the maximum PCI utilization is reduced. The PCI bus is measured to learn how the 21050 transfers data across the bus. When the network card transmits a (large) packet, the 21050 transfers data in chunks of 256 bytes across the PCI bus to its local memory. The 21050 PCI-to-PCI bridge chip employs a FIFO of this size and it uses it efficiently for data transfers.

When data is received by the network card, it transmits the data in chunks of 64 bytes to memory. Since memory-write operations complete quickly because of the Intel 440FX memory controller's DBX, these relatively small transfers do not cause problems: the maximum rate with which the network card can write data to memory (when only considering the PCI bus speed) is 109.0 MB/s.

4.5 Combined performance

When combined network and disk traffic are measured, insight is given in, in particular, the load on the PCI bus. As is shown in earlier sections, the transfer of data to disk can be as high as 90.0 MB/s and from disk the transfer can be as high as 101.5 MB/s. Further the OPPO network card can run at 135.7 Mb/s. How these devices behave when they are connected into the same machine is unknown.

A PCI measurement showed that the OPPO network card transmits a 1,536 byte large AAL5 packet in 24.5 μs across the PCI bus. During this period it releases the PCI bus quite often and the actual bus time for the transfer is only 14.7 μs . This means that when the network device is used at full speed, approximately 17.0 % of the available PCI bandwidth is used for OPPO PCI transfers.

To transmit data at a rate of 135.7 Mb/s, this line rate worth of data needs to be read from disk. A disk transfer of 64 bytes to memory costs 570 ns for the NCR53c810a disk controller, so disk transfers at 135.7 Mb/s costs approximately 15.8 % of the PCI bandwidth. So, in theory there is enough bandwidth available to saturate three OC-3 ATM networks by using OPPO network cards.

An OPPO transmits 64 bytes in 560 ns to memory. This means that when data is received at 135.7 Mb/s, approximately 15.5 % of the PCI bandwidth is used for network transfers.

To write data to disk at network line rate, 135.7 Mb/s needs to be written to disk, with a PCI speed of 64 bytes per 870 ns for the NCR53c810. This means that to run the disks at network speed, 24.2 % of the PCI bandwidth is used for disk transfers. It means that in theory there is enough bandwidth available to transfer the equivalent of two OC-3 network links.

When NCR53c875 SCSI PCI cards are used, 512 bytes can be transferred to the SCSI controller's FIFOs in 4.26 μs for disk writes and 3.92 μs for disk reads. This means that respectively 14.8 % and 13.6 % of the PCI bandwidth is used to run the disks at network speed and that the machine can (in theory) consume the equivalent of three OC-3 ATM networks.

It is shown by the multi-controller NCR53c875 experiment that beyond an aggregate bandwidth of 80 MB/s the main memory performance becomes a bottleneck. This 80 MB/s roughly resembles the copying of two 135.7 Mb/s streams from disk to memory and from memory to disk, so in reality a single Pentium-Pro based machine must be capable of saturating two OC-3 ATM links; approximately half a 622 Mb/s OC-12 ATM link.

4.6 Summary

To design a mixed-media file system that can sustain a high data throughput, it is important to understand the throughput capabilities of each of the components that reside in machines. This chapter first analyzes and describes some existing (I/O) technologies. In particular, emphasis is placed on disks, disk interfaces, networks, I/O buses, and memory technology because these components play a critical role in a mixed-media file system.

Based on the analysis of the I/O technologies a test machine is constructed that is used for performance experimentation. The test equipment consists of a (cheap) 200 MHz Pentium-Pro based machine. Also, all used I/O cards in the machine are not expensive and can, at least in theory, transfer data at high rates.

The experimentation machine is used for actual performance measurements. The performance of a disk (the Quantum Atlas-II) is measured and the measured performance numbers are analyzed. The reason for this is to find out if disk technology dictates the design of a continuous-media server. As expected, a mixed-media file system needs to read and write large blocks for high performance.

It is important to stream data on the PCI bus. When the interface cards are not initialized well, a large fraction of the available PCI bandwidth is wasted on transaction set-up, memory-read times (when writing to disk) and transaction termination. By carefully initializing the SCSI cards, the PCI bus is used in streaming mode and an aggregate bandwidth of 114.3 MB/s is (in theory) possible.

It is also shown that 33 MHz/32 bits PCI buses combined with the Pentium-Pro processor and Intel 440FX memory controller cannot write data to disk faster than 114.3 MB/s. The remaining 11.6 MB/s are lost on the Pentium-Pro's cache-consistency protocol that is enforced by the memory controller. Fortunately, the cache-consistency protocol limits the bus performance only by 10 %. However, multi-disk experiments with Seagate Cheetahs demonstrated that the read-saturation rate for memory is approximately 72 MB/s.

Actual network performance is measured. It is shown that the test machine can send data nearly at OC-3 ATM network speed (92 % of the maximum line rate). Also, it is shown that the used network cards do not overload the PCI bus and that only 15.8 % of the PCI bandwidth is required to run the network at maximum speed. A simple calculation showed that the test machine should be able to saturate 2 OC-3 ATM networks or half a OC-12 ATM network when the machine is equipped with SCSI cards that use large FIFOs.

The measurements clearly demonstrate that a plain Pentium-Pro based machine is a good candidate for a mixed-media file system. More importantly, no special purpose hardware is required for high speed I/O. All the measured equipment can simply be bought from any vendor (except for the used ATM network card).

Chapter 5

Clockwise

Clockwise (Figure 5.1) is a mixed-file storage system with support for continuous-media data storage and best-effort data storage. The Clockwise hardware architecture resembles the architecture that is shown in Figure 1.1: it organizes primary memory, secondary-storage devices and (possibly) tertiary-storage devices in a single storage hierarchy. Clockwise provides memory scheduling, disk scheduling and data-placement techniques to applications that use the Clockwise storage facilities.

Clockwise's primary interface is a *dynamic partition*. Dynamic partitions are data structures that allow the grouping of (large) ranges of disk blocks from disks into a single logically consecutive storage space much like Loge [26], Logical Disk [25], Veritas' subdivisions [97], CrosStor's Volume Manager [39] or other commercially available disk volume managers. Unlike Veritas' subdivisions and CrosStor's Volume Manager, Clockwise currently cannot add redundancy to stored data.

Dynamic partitions are constructed from large *dynamic partition blocks*. The dynamic partition blocks are large to optimize for sequential and bulky traffic. As is shown in Chapter 4, using large blocks is the only way to use disks efficiently.

Clockwise dynamic partitions can group dynamic partition blocks from multiple disks. The throughput of a dynamic partition that is stored on multiple disks can be higher than that of a single disk, since Clockwise allows disk blocks to be accessed in parallel from multiple disks at the same time. When an application presents a dynamic partition block address, Clockwise translates the address to a disk number and real disk address. However, from an application's standpoint, a dynamic partition provides the same semantics as raw disk partitions. Clockwise may re-arrange the physical layout of a dynamic partition without the storage application noticing.

When a storage application opens a dynamic partition, it can request throughput guarantees for reading and writing the dynamic partition. A real-time nonpreemptive deadline-dynamic disk scheduler admits and schedules disk requests from storage applications that require such guarantees. These guarantees, or QoS settings, are negotiated with and translated by Clockwise into *real-time* scheduling parameters. To calculate these scheduling parameters, Clockwise analyzes the layout of the dynamic partition to obtain worst-case service timings. When an application behaves according to its allocated QoS parameters, Clockwise guarantees that it meets *all* of its real-time application deadlines. The dynamic partitions and the disk scheduler together form the

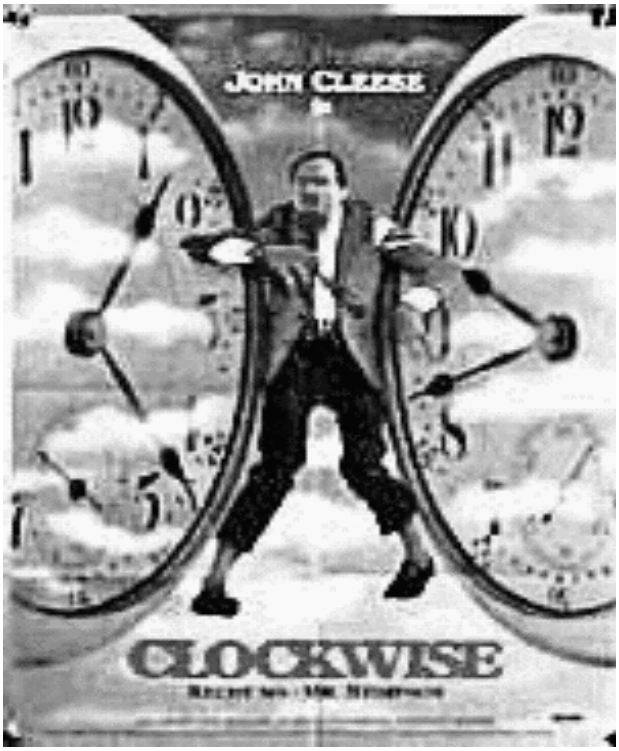

Figure 5.1: Clockwise – the movie [70].

heart of Clockwise: the *Clockwise core*.

The Clockwise core also schedules main memory buffers to allow best-effort file-system applications to cache data and to enable continuous-media applications to buffer continuous-media data. Client applications can allocate a part of the available physical memory as *fixed* or *variable* memory. When an application allocates memory as fixed memory, it can hold on to the memory for as long as the application exists. Applications can temporarily allocate more memory from the variable buffer space to improve performance. A best-effort file-system application, for example, can use this variable memory to enlarge the cache, and continuous-media applications can use this memory for extra read-ahead or write-behind operations. To prevent hogging memory, Clockwise can, and sometimes will, revoke the variable memory space from storage applications.

The Clockwise core is structured such that continuous-media applications that use a Clockwise dynamic partition can almost directly access the underlying hardware for storage and retrieval of continuous-media data. When an application presents a memory address to Clockwise, Clockwise only verifies that the presented memory address references a fixed or variable memory portion for that application. In that case, data is directly DMA-ed in application memory space.

Currently, two types of applications use dynamic partitions for storage: best-effort UNIX file systems and continuous-media real-time applications. These two applications are chosen because it is expected that in the near future best-effort UNIX file systems will need support for continuous-media file storage. This does not imply that there are no other applications that can benefit from Clockwise's capabilities. Applications such as Web servers and real-time databases

can also benefit from the real-time guarantees that are given by Clockwise.

Best-effort (UNIX) file-system applications implement true UNIX file systems inside dynamic partitions on Clockwise. An example of such a UNIX file system is the NetBSD file system tree that is made available for Clockwise. A NetBSD file system simply uses a private dynamic partition for storing the file system. As described earlier, a dynamic partition is organized so that from the outside it behaves *exactly* like an ordinary disk partition; the NetBSD file system does not notice any difference.

A continuous-media application uses a dynamic partition to store or retrieve continuous-media data. When such an application opens a dynamic partition, it requests throughput guarantees that are enough to record or playback an audio and/or video stream. Then, it periodically issues read or write requests to Clockwise to retrieve or store the continuous-media data in a timely manner.

The remainder of this chapter describes each of the Clockwise components in detail. Since Clockwise disk scheduling is a major topic it is not described here, but in a separate chapter (Chapter 6).

5.1 Environmental considerations

Clockwise is developed as part of the ESPRIT Pegasus-I and Pegasus-II projects. The primary goal of these projects is to design an operating system that is capable of treating continuous-media data as a *first class citizen*: applications must be able to manage continuous-media data as easily as they can read, manipulate and write text and numbers.

As part of the Pegasus project, the University of Cambridge developed a new micro-kernel called Nemesis that is able to schedule (continuous-media) applications repetitively and with soft real-time guarantees [37, 64]. Applications can present a desired QoS setting in terms of *period*, *slice* and *latency*, and Nemesis tries to accommodate the application by scheduling the application according to the QoS settings. To make sure the application set is schedulable, Nemesis schedules applications through an *Earliest Deadline First* (EDF) scheduler and it performs an EDF schedulability test [66, 83] before it admits a new QoS setting.

Nemesis goes at great length to prevent *QoS crosstalk*. QoS crosstalk is the phenomenon that when applications use more resources than is agreed upon through the QoS setting, they cannot hinder other (behaving) applications. Nemesis accounts *all* CPU cycles that are made on behalf of an application to the application. Nemesis makes extensive use of shared libraries (called *modules*) to implement operating system services. CPU cycles that are spent in these modules on behalf of an application are accounted to the application [83, 10].

Nemesis modules are called through *closures*. Closures contain state and methods: the methods are the functions of a service and these functions operate on the state that is pointed at by the state part of the closure. A C++ object is an example of a closure. Services that are referred to by a closure can either be local services, in which case the method table points to a shared library, or can refer to a different Nemesis *domain* (*i.e.*, the Nemesis equivalent of a UNIX process), in which case the method table refers to *inter-domain call* (IDC) stubs. A stub compiler called MIDDL generates stubs for both the client and server applications [84]. When a closure

refers to IDC stub methods, only the server side can access the client state.

Nemesis is a *Single Address Space Operating System* (SASOS). This means that all user applications, closures, and services are mapped into a single address space.

5.2 Clockwise core

The Clockwise core is the heart of the storage system. Its primary tasks are the division of the disk and memory resources to the clients that run on top of the Clockwise core. The disk resources can further be subdivided into a number of smaller policies that deal with bandwidth scheduling and placement of data on disks.

All disk-resource scheduling policies are centered around dynamic partitions. These dynamic partitions consists of a number of (preallocated) dynamic partition blocks, of size 1 MB.

Dynamic partition blocks are large because, as is shown in Chapter 4, reading and writing of large blocks is the only way to obtain high disk efficiency. In the same chapter it is also shown that reading or writing multiple tracks is even more efficient. However, given that there can be many different types of disks (with and without zoning) in the storage array, there can also be a large number of different track sizes. From a complexity standpoint it seems more worthwhile to deal with fixed-sized blocks rather than a large number of variable block sizes: Clockwise can reorganize dynamic partitions by moving dynamic partition blocks freely through the storage array for performance reasons. If there are many different block sizes in the storage system, such optimizations may proof difficult to implement.

Presenting many different block sizes to storage applications makes programming the storage server a difficult task. This is in particular true if applications become dependent on the variable block size.

5.2.1 Clockwise disk organization

Figure 5.2 shows a schematic overview of the data layout on Clockwise disks. Each Clockwise disk has a private superblock and a (large) list of free or used dynamic partition blocks. The superblocks contain the logical disk identification of the Clockwise disk and it contains the logical disk address (logical disk number and block address) of a data structure called the *dynamic partition table* (DPTAB). This DPTAB lists for all dynamic partitions the logical disk address and size of the *dynamic partition inode* (DPNODE).[1] Each of the DPNODES contain the logical disk addresses of the allocated dynamic partition blocks. In the figure three different dynamic partitions are indicated.

When Clockwise boots, it reads all of the superblocks from all attached disks and it compares the superblocks. Only when it can find valid Clockwise superblocks from all disks and the system is complete (*i.e.*, there are no missing disks), it proceeds to read in the DPTAB into memory and to analyze which of the dynamic partition blocks are used for storage. Clockwise does this by

[1] An *inode* is a data structure that is used in the UNIX kernel that describes a file's meta-data (including the disk addresses).

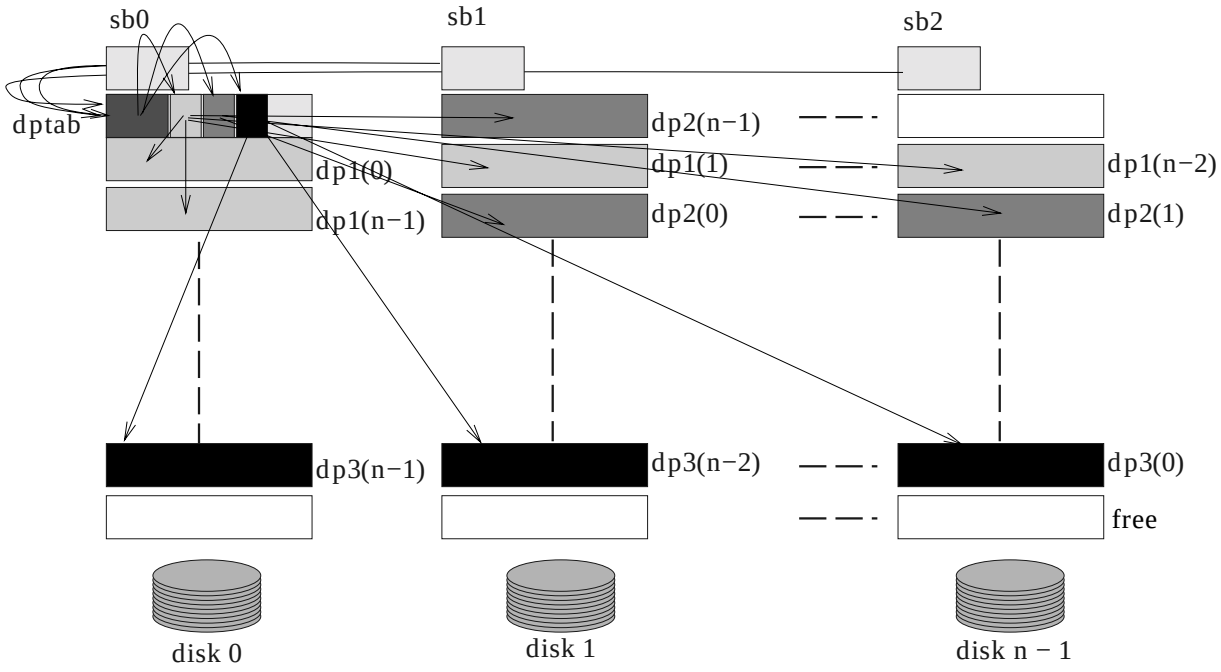

Figure 5.2: Dynamic partition layout on disks. SB0 ... SB2 represents a superblock on disks 0 ... 2, DPTAB represents the dynamic partition table, DP0 ... DP3 represent 3 dynamic partitions.

marking entire dynamic partition blocks used in an in-core bitmap of used blocks when blocks are used to store the DPTAB, a DPNODE or a user dynamic partition block.

The dynamic partition blocks that are used to store the DPTAB or DPNODES are special *meta-blocks* (*e.g.* dynamic partition block 0 of disk 0 in Figure 5.2). Meta-blocks are organized in a log-structured fashion much like LFS [85]. Whenever a new DPNODE is written, it is appended to the meta-data log. If a meta-data block is full, a new meta-data block is allocated from the list of free dynamic partition blocks. Meta-data blocks are occasionally garbage collected by writing all of the meta data to new dynamic partition blocks. Since the update rate to meta-data blocks is quite low, this operation happens only seldom. Also, the amount of meta data is small (typically a few megabytes), so rewriting all of the meta data to disk only costs a few hundred milliseconds.

Updates to the superblocks and DPTAB are rare. The superblocks are only updated when the address of the DPTAB changes or when disks are attached or removed from the array. The address of the DPTAB only changes when the existing DPTAB is too small to list all of the dynamic partitions that are stored in the disk array. The DPTAB itself is changed when a dynamic partition changes (*i.e.*, when a new DPNODE is written to a new location in the meta-data blocks). This happens whenever a new dynamic partition is created, a dynamic partition shrinks or grows, or when the dynamic partition blocks are moved to different disk locations to optimize for performance reasons. Although the DPNODEs are maintained in a log-structured fashion in the meta-data blocks, the DPTAB is changed in place for simplicity.

The superblocks, the DPTAB and the dynamic partition DPNODEs are the only disk data struc-

tures that are used by Clockwise on disk. All other data structures, such as the bitmap of free blocks, and the allocation of data blocks in the meta-block are reconstructed when Clockwise boots.

5.2.2 Using dynamic partitions

Dynamic partitions are presented to storage applications as (possibly long) sequences of consecutive disk blocks. A storage application that runs on top of Clockwise can address a dynamic partition just like it addresses raw disk partitions: it presents a block number and the number of blocks to transfer. Clockwise allows storage applications to address Clockwise dynamic partitions with any block size that is a multiple of the disk's sector size. When an application presents a block number and block size to Clockwise, Clockwise translates the logical-disk address to the real disk number and disk address by means of the dynamic-partition's DPNODE and issues the request to the physical disk.

Blocks that are logically consecutive in a dynamic partition do not have to be stored consecutively in reality. In fact, two consecutive disk blocks do not even have to share a disk. The Clockwise core only guarantees that blocks within a dynamic partition block are stored consecutively.

Client storage applications initiate storage activities by opening a dynamic partition. On an open request, the Clockwise core reads the dynamic partition DPNODE into memory and it creates server state for the storage application. This state contains all information regarding QoS parameters, expected access patterns and, when applicable, the requested data-placement policy.

The interface to read and write data from Clockwise is an asynchronous one: storage applications first queue requests asynchronously in the Clockwise core and can poll or wait for the completion results through another Clockwise function. The reason for this interface is that it fits capture and rendering of continuous-media data: only a single thread of control is needed to keep the disks and network busy. Also, if a system provides an asynchronous interface, it is quite simple to emulate a synchronous interface on top of the asynchronous interface. The other way around is much more difficult.

5.2.3 Data-placement techniques

Storage applications can indicate what kind of dynamic partition block allocation technique they require when creating a new dynamic partition. Storage applications can create or extend a dynamic partition and specify on which disks the dynamic partition blocks need to be allocated, how many of these blocks need to be allocated consecutively and where on disk the blocks need to be allocated. To allow storage applications to determine these allocation parameters, an application can inspect a dynamic-partition's DPNODE. A DPNODE is presented to the storage applications as a variably sized list of tuples with a logical disk identification and a physical block address.

When a storage application does not specify a block allocation policy, Clockwise allocates dynamic partition blocks in a rotating, first-fit fashion. The reason for rotating the dynamic partition blocks across all disks is that when a large bandwidth is required all available disks can

be used in parallel for storing or retrieving the data in the dynamic partitions. This allocations policy also avoids disk *hot spots*.

The default allocation policy may not yield optimal storage facilities for all storage applications that run on Clockwise. As is shown later, using all of the disks in parallel for transfers of, for example, continuous-media data requires large amounts of memory space. To limit memory consumption, applications can reorganize the dynamic partition afterwards. To perform such operations, the storage application informs Clockwise which dynamic partition block needs to be re-mapped to which other location. Clockwise tries to allocate the requested block, and when the allocation succeeds, it moves the contents of the old dynamic partition block to the newly allocated dynamic partition block and updates the dynamic-partition's DPNODE.

Clockwise itself does not store any redundant information on dynamic partition blocks to guard against disk failures. If a storage application requires storage strategies such as RAID [18, 76], mirroring [104], or *Random Data Assignment* (RDA) [56], it is up to the application to assign blocks to disks and read or write the redundant information.

5.2.4 Memory management policies

When continuous-media data storage and best-effort data storage are combined in a single server, this server needs to know how to distribute the memory resources to the storage applications. Continuous-media applications usually use memory as an intermediary step before data is sent to the network or to disk and best-effort applications, such as UNIX file systems, use the memory to cache disk blocks.

If a storage system guarantees real-time service, as is done by Clockwise, continuous-media applications only require two buffers to play back or record continuous-media data. When playing back data, Clockwise uses one buffer to fill it up with continuous-media data while the other buffer is used to play data onto the network. If there is large variation in the load, a continuous-media application may need to preload some more buffers or use more buffers in a write-behind scheme to smooth disk traffic. In these cases, Clockwise needs to provide extra memory resources.

Best-effort UNIX file-system applications, on the other hand, usually perform better if more memory space is available for them. The extra memory space is used to cache more data in order to achieve higher cache hit rates [6]. Higher cache hit rates imply that fewer operations need to be serviced on disk and the mean latency for file-system operations decreases. Extended caches can also be used for extensive write-behind caching [13], and reduce contention on the I/O queue further.

To combine both application types in a single system, Clockwise needs to be able to mediate memory requests between such clients. To do this, Clockwise has defined two types of memory spaces: *fixed* and *variable* memory space. Fixed buffers are buffers that, once allocated, the application can hold onto. Continuous-media applications use the fixed memory space to implement a double buffering scheme, and best-effort applications such as UNIX file systems can maintain their (dirty) cache in the fixed buffers.

When applications (temporarily) require more memory space, they can allocate more memory from Clockwise in the variable buffer space. When there is enough space available, Clock-

wise hands out this memory through a *First Come First Serve* (FCFS) policy.

If Clockwise is out of memory, it starts balancing the memory requests over all the applications. The Clockwise memory-scheduling policy is that each application needs to be able to use fair share of the available memory. A fair share is defined such that all applications can allocate the same ratio of fixed and variable buffers. When Clockwise is out of memory and new memory requests arrive in Clockwise, Clockwise first determines which applications use more than their fair share of memory and informs those applications to release some memory. For this, each application that uses variable buffers must provide a Nemesis event counter. Nemesis allows two applications to share an event counter; both can read and increase the counter atomically. Clockwise increases the event counter with the amount of buffers that needs to be returned by the application. The storage application polls the event counter and determines the amount of buffers it needs to return by comparing the previous value of the event counter with new values.

When an application needs to return a number of buffers, it can do so within a certain time limit. Currently this time limit is 500 ms from the time the event counter is increased. The reason for this large time frame is that the application buffers can be locked in I/O devices (*e.g.*: network-ring buffers) and it may take some time before the buffers are unlocked again. Also, when the buffers are used to run an UNIX file-system's write cache, the buffers may need to be flushed to disk before the buffers can be released. The CELLO disk trace of the HP Laboratories disk traces (see Section 3.2.2) contains large bursts of requests with maximum request latencies of more than a second. Given that the storage application needs to be able to return data within 500 ms to Clockwise, such a file system must allocate enough fixed buffers to sustain the largest disk bursts. It should only use variable buffers for the non-dirty cache.

If the applications do not release the buffers within the time frame and the memory is still required by competing applications, Clockwise can take the buffers by force from misbehaving applications. For this it calls out to the memory manager to revoke the read and/or write permission on the variable buffers for selected misbehaving applications. Since Clockwise does not know which buffers to repossess, it grabs the first ones it can find. This is not considered a problem since, if the application would have behaved, it would have returned the buffers in an orderly fashion.

When the memory-allocation policy is applied to a system with a number of UNIX file systems that are started at boot time and a number of continuous-media applications, the UNIX file systems quickly divide the memory amongst themselves in fixed and variable portions. When continuous-media applications enter the system, they ask Clockwise for some memory, which, on its turn, calls to the best-effort applications – *e.g.* the file systems – to free up some memory. When enough buffers are freed up by the file systems the continuous-media applications can start sending or receiving continuous-media data.

When continuous-media applications require more memory to deal with load spikes, they can call to Clockwise, but Clockwise might not return the buffers instantaneously – other applications have 500 ms to respond to the request. This may imply that the continuous-media applications do not have enough buffers to record or play back the continuous-media data and they can loose video or audio fragments. To prevent excessive continuous-media data loss, Clockwise always maintains a small portion of free memory to serve requests quickly. However, there is a chance that there is simply not enough memory quickly available. This is not considered a problem

because the continuous-media application is not running within its negotiated memory QoS parameters in the first place.

When an application terminates, the memory held by the application is freed up again. When this event is noticed by the other best-effort applications, they may redistribute the freed up memory amongst themselves. Typically a best-effort file system periodically polls Clockwise to obtain more variable buffer space.

5.2.5 Efficient I/O

To support efficient I/O between devices in a Clockwise machine, Clockwise needs to be able to store and retrieve data to and from disk without CPU data copying as is shown in Chapter 4. In reality this means that Clockwise needs to provide a direct interface to the DMA engine of the SCSI cards.

To complicate matters, Clockwise's DMA interface needs to be able to store and retrieve data in or from non-consecutive memory buffers. Consider, for example, the storage of a continuous-media file. If such data arrives at a Clockwise machine through some sort of network, usually the payload (*i.e.*, digitized audio or video) is packed with a packet header and a trailer. It is likely that only the payload needs to be written to disk and to avoid wasting disk resources, payloads will need to be packed back-to-back on disk. To avoid CPU copying, Clockwise must be capable of only extracting the payload from the network packet through the SCSI card's DMA engine. A similar situation occurs for network transmissions: data that is read from disk needs to be stored in already prepared network headers and trailers.

When the SCSI DMA engine copies data directly from and to user buffers, Clockwise needs to make sure that the presented buffer addresses are valid addresses. This means that Clockwise needs to verify that the presented addresses really belong to the application and can be read from or written to. Clockwise does this by only allowing data addresses that point to earlier allocated fixed or variable buffers for data transfers. Whenever an application requests work, Clockwise verifies that the data buffers are part of the preallocated set of data buffers for that application. This way it is not possible that a DMA engine is loaded with incorrect data pointers and data corruption by other applications cannot occur.

Most SCSI cards support scatter-gather lists. A scatter-gather list is a list of pointers to buffers that may or may not be stored consecutively in memory. When such a list is presented to a DMA engine, it extracts or inserts the data in a (possibly) non-contiguous manner from or to memory.

The Clockwise interface is extended with this scatter-gather interface. A Clockwise application can present the data that needs to be sent to or received from disk as a (possibly) long list of data pointers. After verification of the data pointers, Clockwise copies the pointer list into the SCSI DMA engine. An application that needs to strip network packet header and trailers, informs Clockwise of the location and size of the payloads within the packets and leave the extraction of data to the SCSI DMA device.

For network transmission, the only operation an application needs to perform is to prepare a list of network headers and trailers and inform Clockwise where to insert the data in the packets. In this case, when Clockwise finishes a scatter-gather disk-read operation, the application

only needs to inform the network device of the location of the consecutive network packet for transmission.

5.3 Best-effort applications

There can be many best-effort applications that make use of the Clockwise dynamic partitions. Best-effort applications can be entire database systems, UNIX file systems, news applications or Web applications (*e.g.*: Web servers).

Of particular interest is the storage and retrieval of UNIX data on UNIX file systems. The reasons are pragmatic:

- Since the source code of UNIX file systems is available, it is relatively straightforward to make these systems available under Clockwise.

- UNIX file systems are in use on a daily basis in the Huygens laboratory by a large number of people. When such a file system is available under Clockwise, realistic performance measurements and experiments can take place.

- Since UNIX file systems are in use by many scientific groups throughout the world, results obtained by Clockwise can be validated and reproduced by others.

There are many UNIX file systems today, and each optimizes for different load characteristics (see also Chapter 2). McKusick's Fast File System (FFS) [71] optimizes by clustering groups of cylinders, and files that are located in the same user directory are stored in the same cylinder group. If a directory is read, the disk heads are already located at the correct cylinder for reading a file from that directory. Rosenblum's Log-structured File System (LFS) [85] recognizes that by using large read caches, disks are dominantly used for disk-write operations. By optimizing disk-write accesses to disk, the entire file system is faster. There exist several other UNIX file systems that optimize for yet other load characteristics, such as Seltzer's EFS [91] and Linux' EXT2FS.

It is a tremendous task to make available each brand of UNIX file system under Clockwise. Fortunately, the NetBSD[2] UNIX operating system implements a *virtual-file system* (VFS) [55] that has incorporated a few UNIX file systems. Figure 5.3 shows the layout of the NetBSD VFS in the box 'standard VFS'. There exist a top level library that provides a Posix-like interface [38], which maps the file-system independent calls to specific file-system implementation calls. The bottom layer of VFS normally interfaces directly to raw disk partitions through UNIX device drivers. By making VFS available under Clockwise, each of the incorporated file systems is also available under Clockwise. Currently the following file systems are incorporated in VFS: McKusick's FFS, Linux' EXT2FS, a Memory FS (MFS), a Union FS (union) and an implementation of BSD LFS [89].

[2] NetBSD is a public domain implementation of the Berkeley BSD4.4 system.

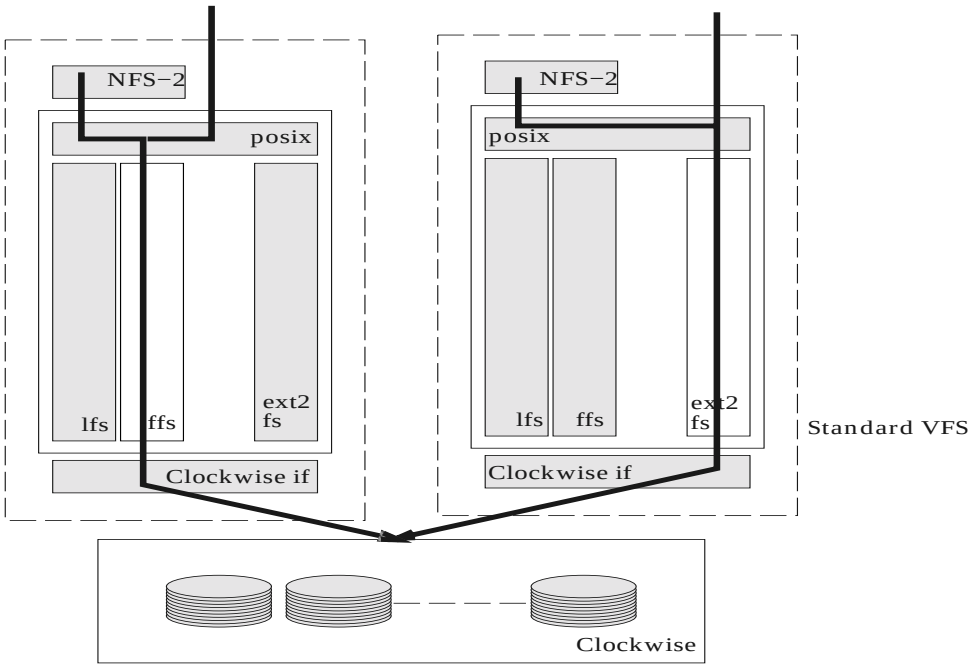

Figure 5.3: Virtual File-System organization.

Porting VFS to Clockwise/Nemesis

Porting VFS to Clockwise/Nemesis means that the UNIX VFS kernel implementation needs to be ported to Nemesis. Since the UNIX VFS implementation is a large and complicated system, it was decided to implement the entire VFS tree as a single Nemesis service, called the VFS server. No effort was put in breaking up VFS into smaller Nemesis modules and to account CPU usage on a per application basis, much like what is done for $\mathcal{EFS}$ [10, 9].

The first step in porting VFS to Nemesis is that the VFS POSIX interface needs to be made available to clients that run on the Clockwise machine and VFS needs an interface to the Clockwise core. By gluing the POSIX interface into Nemesis, all of the applications that run on the Clockwise machine can use UNIX-like storage facilities through Nemesis IDC. The interface to the Clockwise core is provided by a UNIX-VFS device driver that, instead of communicating with a UNIX device driver, uses the Clockwise core interface to store and retrieve data blocks from dynamic partitions. VFS also calls a number of UNIX kernel primitives to perform inter-process synchronization, mutual exclusion and user identification. Inter-process synchronization and mutual exclusion are implemented through equivalent Nemesis functionality.

To distinguish between users, VFS requires *user identifications*. However, user identification is not yet implemented on Nemesis. Since Nemesis currently is not a multi-user system, every Nemesis application can access all resources that are implemented on that machine. This means that Nemesis users are the equivalent of a UNIX user that is able to access all resources, *i.e.*

superuser. Therefore, all Nemesis users are simply given the user-id that represents the UNIX superuser.

The VFS tree is also extended with a NFS version-2 interface [45, 88] as is shown in Figure 5.3. This interface maps external NFS requests to internal Clockwise/POSIX calls to execute the NFS operations on a file system. In the NFS protocol, a UNIX user- and group-identification are transmitted as part of the NFS request. This user and group identification is used to perform the request on the file system.

UNIX systems use a *process* structure to record state that is associated with a process. The process structure records a process-id, the open files for that process, the user- and group-id(s), the current working directory and many other fields. Clockwise/Nemesis applications that use VFS initialize this state when they connect to VFS. The state itself is kept in a Nemesis closure. For VFS, the method table of the closure holds the POSIX functions and the state field holds a reference to a replacement UNIX process structure. This process state is initialized when an application first contacts the VFS server. When the application finishes, it unregisters at the VFS server and the server disposes of the client state.

As said, VFS implements a number of different file systems. When VFS boots, the VFS initialization software finds out the type of file system that is stored on disk (or, in the case of Clockwise, in a dynamic partition) by reading the file-system superblock. It then creates a local function array that maps the generic POSIX functions onto file-system specific operations (*e.g.*, `vfs_read` refers `ffs_read`). This function array points to file-system functions in the white area in Figure 5.3.

VFS file-system caching is implemented by a simple LRU-based block cache. Since Clockwise also implements memory scheduling and only allows I/Os from its own memory blocks, VFS allocates blocks from Clockwise rather than from the underlying operating system. When a VFS is started, it allocates a fixed memory portion and as much variable memory as possible to keep the block cache. When Clockwise needs memory for other applications, VFS can be forced to return the variable portion of the memory to Clockwise. For this, VFS implements a separate thread of control that waits for memory requests that are made by Clockwise on memory event counter. When the thread is activated, VFS first tries to return as many free buffers as possible before returning non-dirty and later dirty buffers (the dirty buffers are first flushed to disk).

Since there can be many VFSs that run on Clockwise, each VFS is implemented as a separate Nemesis process with its own CPU QoS settings. Each instantiation of a VFS accesses its own private Clockwise dynamic partition. In Figure 5.3 two VFSs are shown: one that implements a FFS in a dynamic partition and one that implements an EXT2FS in another dynamic partition. In the example, the LFS file system is not used.

The private CPU QoS settings per VFS protects instantiations from resource misuse of other instantiations or Nemesis processes that run on the Clockwise server. Each VFS also comes with a private, *i.e.* per process, NFS implementation, and because the Ethernet and IP stack processing are also explicitly scheduled in Nemesis [11], one Nemesis (VFS or any other) application simply cannot influence the performance of other VFSs. Since each VFS has its own NFS server, remote clients need to bind to a particular NFS server on the Clockwise machine.[3]

[3]Only one of the servers listens to UDP port 2049.

Each VFS file system is registered under its own file-system name in Nemesis. When a Nemesis application wants to use a particular file system, it binds to the file system by using its file-system name. The file-system name is identical to Clockwise's dynamic partition name.

Two auxiliary programs are available on the Clockwise server to allow remote-client applications to NFS-mount a VFS file system that runs on Clockwise. The Nemesis port mapper allows remote clients to find the UDP port to which the Nemesis NFS-mount daemon is listening and the Nemesis NFS-mount daemon is used to find the root NFS file handle in a UNIX VFS. This root file handle is used by the client as a starting point for the remote file system and is embedded into the client machine's operating system name space.

Currently, only a single VFS file-system type is actually tested (McKusick's FFS). The reason for this is that it is not trivial to port the file-system helper programs to Nemesis because each helper program's disk interface needs to be replaced with a Clockwise interface. A Posix compliant user library for Clockwise would simplify porting matters considerably. The helper programs are *newfs(8)*, a utility to create a new file system on a Clockwise dynamic partition and *fsck(8)*, a utility to repair an inconsistent file system after a crash of the host operating system.

5.4 Continuous-media applications

Continuous-media applications use Clockwise to store and retrieve continuous-media data in or from dynamic partitions. The bulk data is stored in-band in dynamic partitions. The index information for the continuous-media data is stored out-of-band in, for example, a VFS file system, so that the index can be examined without having to access the raw continuous-media data.

Continuous-media applications are activated and controlled by a continuous-media session manager that runs as a separate process on the Clockwise server machine as is shown in Figure 5.4. In this figure, a (remote) client workstation, shown by the UNIX workstation sends requests to the Clockwise continuous-media session manager. This session manager can start, alter or stop continuous-media applications. In the figure only playback sessions are shown and all of them are controlled through the continuous-media session manager. Each of the continuous-media applications shown in the figure sends continuous-media data on an ATM virtual circuit to a rendering device, such as an ATV (see Section 3.1.1). In the figure the 'Wallace and Grommit: A Close Shave' audio and video streams are sent to the ATV. The audio track 'Kinderpindakaas' is sent to another device over a different ATM virtual circuit.

To start a continuous-media application, a remote client sends a request to the session manager to start a particular continuous-media application on the Clockwise machine. For this, the remote client specifies the name of the application, the name of the dynamic partition to use, the names of the index files and the (memory and disk) QoS parameters. The session manager creates a new Nemesis domain and starts the requested application. When there are insufficient resources available, the session manager cleans up the already allocated resources and informs the remote client of the failure.

During the playback or recording, a remote client can alter the parameters of the playback or recording. To alter session parameters, a client sends a message to the session manager, which, on its turn, forwards the parameters to the continuous-media application by means of standard

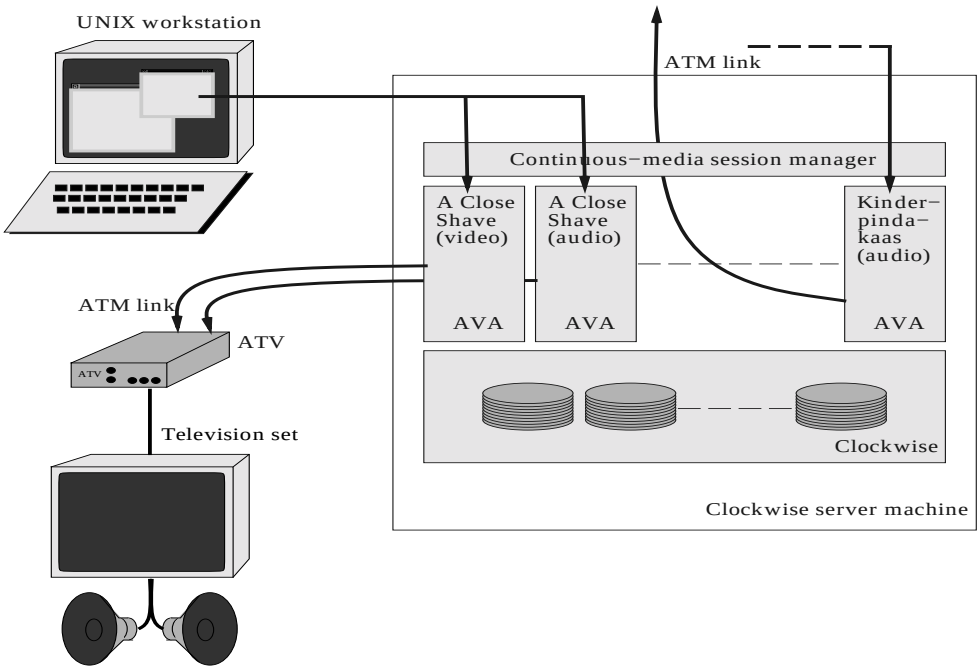

Figure 5.4: The Clockwise continuous-media system.

Nemesis IDC primitives.

To keep a recording or play back active, remote clients periodically send a message to the session manager. When the remote client dies, the recording or playback also automatically finishes. This is especially important when recording a stream: this is the only reasonable way to determine that a recording should stop.

When a remote client indicates that it would like to finish the session, or when it has not sent a message to the session manager for some time, the session manager terminates the continuous-media application.

The Clockwise continuous-media applications themselves are in principle simple applications: they exchange continuous-media data between network and disk. As is described in Section 5.2.5, Clockwise offers a direct data path to the disk hardware and enables data movement without the CPU having to copy data. This means that the only operations continuous-media applications need to perform are the recording or interpreting of continuous-media index files to decode independent continuous-media fragments, managing preload or write-behind I/O buffers, to prepare (scatter-gather) disk requests, and (possibly) synchronizing the streams with other streams in the system.

One of the continuous-media applications that can be started through the continuous-media session manager is the AVA continuous-media application. This application records and plays back digitized audio and video from AVAs (see Section 3.1.1).

Other digitizing hardware usually consists of host-interface cards to which a camera, a micro-

phone or speakers can be connected. These cards digitize the received data and DMA the data into host memory for further processing [63]. These cards are, from a technical perspective, similar the AVAs, with one subtle though important difference: cameras, microphones and televisions can be connected to a system without a requiring a nearby computer. As long as there is an ATM network, audio and video streams can be transmitted.

5.4.1 QoS parameters

Nemesis requires CPU QoS parameters to activate a continuous-media application domain. A remote client sends the CPU QoS to the session manager, which, on its turn, presents these QoS parameters to Nemesis. The CPU QoS parameters are the aforementioned *period*, *slice* and *latency* and determine how Nemesis schedules the new continuous-media application domain. If, for example, a 30 frame per second NTSC video is played, a likely value for the period parameter is 33 ms so that the application runs whenever a new frame needs to be transmitted. The slice parameter determines how long the application needs to run during a period. The duration of the slice depends on the continuous-media application itself. The latency parameter depends on the sensitivity of the user – the less sensitive the user is to jitter in the audio or video stream, the larger this value can be. Larger values for latency allow the CPU scheduler to schedule the tasks with more freedom [83].

When a continuous-media application is activated through the continuous-media session manager, it can also be provided with a set of network QoS parameters. These parameters are used to negotiate a service contract with the network device driver. The network QoS determines how many bytes are guaranteed to arrive or to depart through the network interface and, as can be the case in an ATM network, values like *Peak Cell Rates* (PCR) can be supplied. If a network does not have support for QoS, this parameter is void.[4]

The (remote) client also provides the continuous-media application with disk and memory QoS parameters. The disk parameters determine how Clockwise schedules disk requests from the application and the memory parameters determines how many memory buffers are allocated from Clockwise. When the continuous-media application is used to record or playback a constant bit rate stream, calculating the disk and memory QoS parameters are simple matters. The (remote) client determines a block size that it likes to use for the recording or playback and it determines the required bandwidth based on the actual bit rate. Since Clockwise always meets its disk deadlines when the application behaves according to its negotiated QoS contract (as is shown in Chapter 6) only two memory buffers with the size of the user-block size are required.

Variable bit rate calculations are more complicated. The bit rate for playback can be calculated exactly by inspecting the index data, but when a stream is recorded with unknown compression rates, the bit rate can only be guessed. The bit rate and memory requirements can be underestimated in the hope that during the session *enough* spare resources are available to guarantee smooth playback or recording.

During the session, a (remote) client can instruct the continuous-media application to renegotiate its QoS contract with Clockwise. When the continuous-media application has allocated

[4] Which is currently the case.

too few buffers or too little disk bandwidth, it can renegotiate new memory QoS setting or disk parameters.

5.4.2 AVA recordings

The AVA continuous-media applications on Clockwise record or playback AVA audio and video streams. They communicate with an ATM network card and Clockwise to store and retrieve digitized AVA audio and video to and from disk. The primary goal of the continuous-media application is to gather enough ATM AAL5 packets from an AVA to fill up a Clockwise dynamic partition block, or to read such a block from disk and to prepare it for transmission across the ATM network.

Figure 5.5 shows the workings of an AVA continuous-media application in detail. The top continuous-media application records data, the bottom plays back data. Light blocks are empty buffers, dark blocks are full buffers. Continuous-media data first arrives in the Clockwise server machine through a Digital ATM Works 350 (a.k.a. OPPO – See Section 4.4) and is stored in buffers that are preloaded in a ring buffer. When a buffer is filled up with a network packet, the OPPO device driver's interrupt routine informs the application of this event. The AVA continuous-media application picks up the buffer descriptors from the OPPO card and when enough data is gathered to fill up an entire dynamic partition block, the AVA continuous-media application sends an array of network buffer descriptors as a scatter-gather list to Clockwise. When data is written to disk, the empty buffers are recycled to the network card.

Transmission of audio and video works similarly: the AVA continuous-media application loads a dynamic partition block from Clockwise in such a way that it can directly forward the independent AAL5 packets from the block to the OPPO card. When the packets are transmitted, the device driver's interrupt routine signals the application to free up the buffers. Afterwards, the AVA continuous-media application can use the buffers again for loading data from disk.

When video data is compressed by Motion-JPEG compression, the AAL5 packets can be quite small. Consider for example a configuration where the 768×288 PAL television frame is transmitted as 72 AAL5 packets – a common configuration for the AVA – to a Clockwise server. This means that 384×8 pixels are transmitted per AAL5 packet, with an uncompressed size of approximately 3 KB. If, however, Motion-JPEG is enabled, the resulting packet size can be any size between 200 bytes and 3 KB.

The network driver only operates with fixed-sized buffers, and it can only receive an entire network packet if the preloaded network buffer is large enough. If too small buffers are used, packet ends are discarded by the interface card. This implies that the application needs to prepare for the worst-case scenario when a stream is recorded. Not all packets compress equally well, some may not compress at all. The ring buffer is loaded with buffers of 4 KB, which is large enough to hold the worst-case AAL5 packet. Given that a continuous-media application uses large Clockwise dynamic partition blocks for the recording of continuous-media data and the best compression leads to packets of 200 bytes, a total of 15 MB worth of worst-case network packets need to be gathered before a single megabyte is filled. Since double buffering is used, 30 MB of memory per stream can be used in the worst case. This is clearly undesirable.

By altering the device driver, it is now able the re-use data buffers that are not used up fully

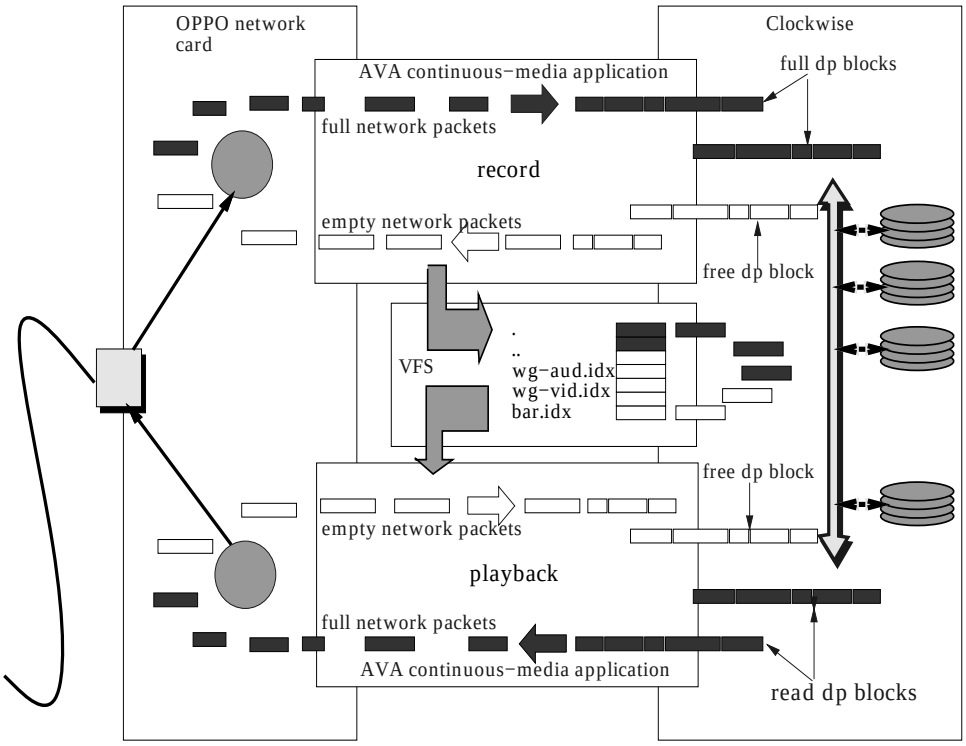

Figure 5.5: Audio and video transmission.

when a stream is recorded. So, instead of loading a number of buffers into the device driver to hold a single AAL5 packet, larger buffers are loaded into the device driver to hold a number of AAL5 packets. When an AAL5 packet is received into such a larger buffer and there is ample room to receive another AAL5 packet with the maximum size into the same buffer, the remainder of the larger buffer is re-used and is reloaded into the ring buffer. If 64 KB buffers are used, the maximum fragmentation is 6 %.

5.4.3 AVA index information

Figure 5.5 shows that there is an interaction between the AVA continuous-media applications and a VFS file-system. The VFS file systems are used to store and retrieve index information that is needed to find individual frames and AAL5 packets in the stored dynamic partition blocks. Without index information, earlier recorded raw (and possibly compressed) audio and video cannot be decoded into independent frames of information again.

In general there can be many layers of indexing information available for audio and video streams. These indexing types range from the actual pointers to raw data such as is the case for AVA stream-index information to high-level multimedia indexing information [33]. However, without the index information to the raw audio and video, high-level multimedia editors and player cannot be built.

AVAs do not transmit independent streams of index data. This means that to obtain timing

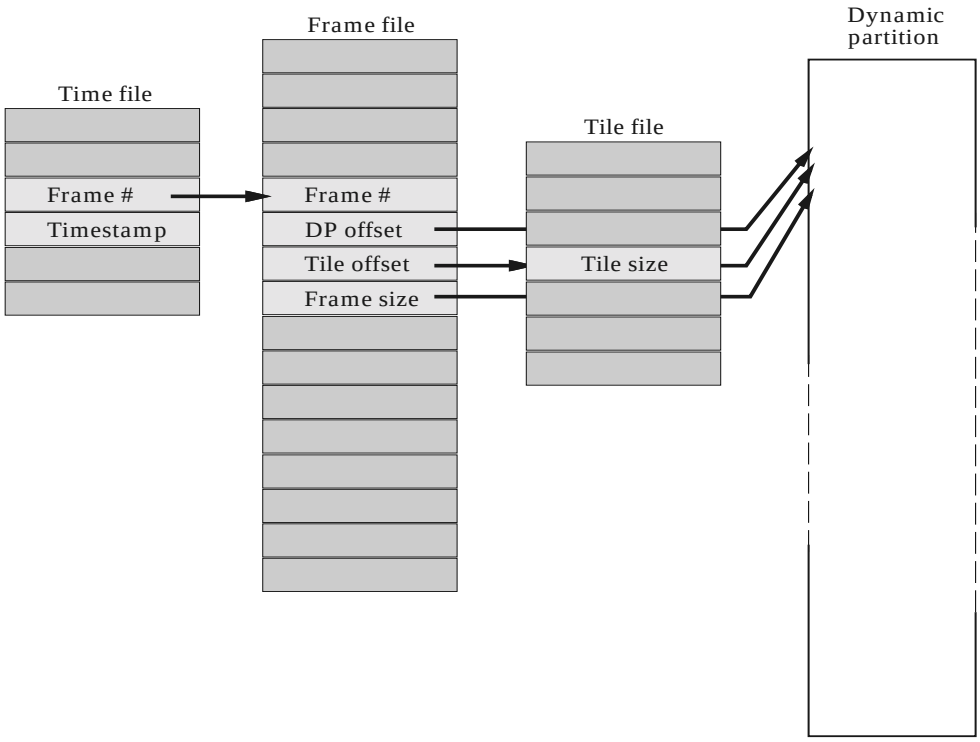

Figure 5.6: AVA index information.

information and frame information, the AVA continuous-media application needs to examine each received AAL5 packet to obtain the meta-data. Each AAL5 packet is examined for the packet size, the frame number to which the packet belongs and the index of the packet within the frame. Furthermore, the time of arrival is recorded to construct a time-index file that is used to associate frame numbers with relative times since the start of the stream.

In total, three files per AVA recording are kept: a *time*, *tile* and *frame* file. The time file records the relative time of a frame within the movie. The tile file records the sizes of independent packets in a frame. The frame file records the byte offsets in the dynamic partition of independent frames within the movie and it specifies where in the tile file the frame starts. When an AVA stream is recorded, the AVA continuous-media application generates these index files on the fly. The index files are shown in Figure 5.6.

The tile-index file records the sizes of the independent AAL5 packets within the continuous-media dynamic partition. The reason for recording the independent packet sizes is because video compression leads to variably sized AAL5 packets. If no compression is used, all packets are of equal size and the tile index file can be discarded. However, since video and audio editors may coalesce streams with different packet sizes, keeping the tile file simplifies playbacks.

The frame-index file holds for all recorded frames (audio or video) the frame number, the offset of the frame (in bytes) within the Clockwise dynamic partition, the offset of the frame in the tile-index file and the size of the frame. This file can be used to play any particular range of video or audio: by examining the dynamic partition offset, the raw data can be found and through

the tile-index file offset, the sizes of the independent tiles can be found. The size field can be used to quickly step through the video or audio file.

To playback a range of frames, an application first examines the frame file to find the offsets of the frames within the dynamic partition and issues read requests for the dynamic partition blocks to Clockwise. When the dynamic partition blocks are read, the application reads the sizes of the tiles through the tile file and sends the independent AVA packets to, for example, an ATV.

Timing information is generated whenever the AVA continuous-media application receives an AAL5 packet. The AVA continuous-media application records the Clockwise server's absolute time for every received message. The AVA continuous-media application generates an entry in the time file for every Clockwise dynamic-partition block that is written to disk. The reason for only recording a single entry per dynamic partition block is that all packets within a frame have the same or almost the same receive time – they are all part of the same frame of video – which means that only recording a single time field per frame is good enough. Secondly, if the frame rate is known, the playback time of a frame can simply be calculated through the previous frame. This implies that, in theory, there does not need to be a time file at all. But frame rates are never known precisely because the clocks in participating devices always drift to some extent. If multiple audio and video streams are recorded simultaneously, and they need to be played back synchronously, a single clock should supply the timing information. This clock is normally that of Clockwise and it is encoded in the time file.

When the latency between the AVA and Clockwise is long or there is a significant jitter, the timing information provided by Clockwise can be inaccurate. In this case it is important to supply the AVA continuous-media application with a separate timing source. This time source can be a machine close to the AVA. The AVA is configured to send one frame per second to the timing machine. The timing machine constructs a time file and sends this to Clockwise.[5]

The index files themselves are stored on VFS. In some sense the index files themselves can also be considered as continuous-media data: for every tile, frame or Clockwise dynamic partition block, the AVA continuous-media application outputs or examines the next part of an index file. VFS, however, does not implement real-time streams. When continuity for index streams is important – for example, when the system is heavily loaded with real-time streams and there is no slack time available – a private dynamic partition can be created for the index files. This dynamic partition can henceforth be scheduled with real-time guarantees. Alternatively, the index files can be buffered in memory.

5.4.4 AVA stream synchronization

AVAs send their audio and video files in separate streams. This makes it easier to process them separately, but it does necessitate a synchronization mechanism.

Each AVA continuous-media application is instructed to receive AAL5 packets from one virtual circuit only and to write the received AAL5 packets to a private Clockwise dynamic partition. The advantage of using separate audio and video dynamic partitions is that it is much simpler to

[5] A similar set up was used by Ian Pratt of the University of Cambridge to demonstrate audio synchronization at the Pegasus workshop in 1996.

play-back the audio and video separately. Although it is possible to construct an AVA continuous-media application that receives data on multiple virtual circuits at the same time, this complicates the structure of the AVA continuous-media application considerably. By storing audio and video separately with separate instantiations of the AVA continuous-media application, both streams can have a different set of QoS parameters thereby reducing the complexity of constructing combined audio and video QoS settings.

The disadvantage of recording the audio and video streams separately is that streams need to be synchronized when they are played back simultaneously. Since an absolute time track does not exist – the AVA does not produce such a stream – synchronization can only be performed through the earlier described time files. When streams are recorded simultaneously, their time tracks are the same and correlating the streams at a later instance is possible.

When a playback session is started with a number of video and/or audio streams, one of the streams is selected as the master stream. All other AVA continuous-media applications periodically adjust their internal timing to the timing information provided by the master stream.

The master stream structure has proven to be good enough for *lip-synchronization*. Audio and video streams need to be synchronized to approximately 80 ms to be considered lip-synchronized.[6] In reality the synchronization is much better than the 80 ms: time differences between streams of at most 1 ms are not uncommon.

5.5 Summary

This chapter presents Clockwise, a mixed-media file system. An overview is given of the Clockwise core, a server that distributes the memory and disk resources to a number of storage applications that run on the Clockwise server machine. The Clockwise core allows storage applications to present memory and disk requirements in the form of QoS contracts. The memory resource is distributed fairly to applications that run on the Clockwise core and the Clockwise core prevents memory QoS crosstalk. The disk resource is scheduled by means of a real-time disk scheduler. However, since disk scheduling is a large topic, it is described separately in Chapter 6.

The Clockwise core has organized all secondary storage in dynamic partitions. A dynamic partition is a data structure that groups large dynamic partitions blocks from all disks into a single logical structure. This logical structure is presented to the storage applications as a raw disk partition. Storage applications can access the blocks as if they are stored consecutively on a single disk. Dynamic partitions can grow and shrink in size and they can be moved around on the disks without the client application noticing. The prime advantage is that disks can be used in parallel without requiring changes to the storage applications and that, for optimization purposes, the layout of dynamic partitions can be changed *after* they are created.

The Clockwise core I/O interface is constructed such that storage applications can access the underlying disk hardware almost directly and that these applications do not have to copy data to send or receive data from disk. This means that the applications that run on a Clockwise core can benefit from the full aggregate I/O bandwidth of the underlying disk hardware.

[6] From Paul Sijben's draft PhD thesis. More information on synchronization can be found in [95].

The two types of applications that are available under Clockwise/Nemesis are UNIX file systems, based on NetBSD's *Virtual File System* (VFS) and AVA continuous-media applications. VFS implements a range of file systems that each optimize for a different set of UNIX file-system access patterns. By making available the entire VFS system under Clockwise, all of the file systems that are incorporated in VFS are also available for storage purposes under Nemesis. An instantiation of a VFS file system uses a Clockwise dynamic partition as back-end for storage.

The continuous-media application that is available under Clockwise allows the recording and playback of AVA audio and video data. An AVA is a device that digitizes audio or video and sends the digitized audio or video on an ATM virtual circuit. The AVA continuous-media application is able to receive the digitized audio or video fragments and to store them in a dynamic partition. The AVA continuous-media application can send the data over another ATM virtual circuit either to an ATV, a device that receives audio and/or video fragments and renders them on a television set, or to a remote workstation for rendering. The AVA continuous-media application interacts with best-effort file systems to store and retrieve index files for the continuous-media data, it uses the scatter-gather facilities of Clockwise to allow zero-CPU-copy storage and playback and it communicates with Clockwise to allow streams to be synchronized with each other.

Chapter 6

Disk scheduling

The heart of Clockwise is the disk scheduler because it provides much of the QoS support. This scheduler determines in which order the requests are sent to disk. It plays an important role to determine the *responsiveness* of the system for best-effort tasks and in the effectiveness of meeting the deadlines for the continuous-media tasks.

The primary goal of the Clockwise disk scheduler is to give priority to best-effort requests if real-time request deadlines are not endangered. The reason why this is important is not hard to see. If real-time requests are always given priority over best-effort requests, the latency of best-effort requests quickly increases when the number of real-time tasks increase. As is described in Chapter 4, disk requests need to be large for high throughput. Given that Clockwise dynamic partition blocks can be as large as a megabyte, a best-effort request can delayed by the number of continuous-media streams multiplied with the transfer time for this megabyte block (between 115 and 184 ms per megabyte on a Quantum Atlas-II disk). By giving priority to best-effort requests, best-effort requests complete earlier and real-time requests complete closer to their deadlines.

The technique of giving precedence to best-effort requests is not new. The Symphony system [93], for example, uses a technique to calculate the *Latest Start Time* (LST) of real-time requests. When real-time requests are queued, Symphony's *Earliest Deadline First* (EDF) scheduler calculates the time at which a real-time request needs to be started at the latest to guarantee the deadline is still met. Whenever there is a best-effort request available, the Symphony disk scheduler gives precedence to the best-effort request if the latest start time of the real-time task is not yet reached. However, a close examination of Symphony's scheduling technique reveals that Symphony does not always meet its deadlines. This is shown later in this chapter.

The Symphony technique guarantees low best-effort latencies, at the expense of a complex scheduler that calculates the latest start time of requests. To insert a request into a queue, Symphony first determines the position of the request in the EDF queue. Next, it recalculates the latest start times of the requests that precede the newly queued request since the request insertion may have changed the latest start times of those requests.

In this chapter a new EDF scheduling technique is presented that is used by Clockwise to schedule the disks with the same objectives as Symphony: best-effort requests are serviced as quickly as is possible and real-time requests always meet their deadlines. The Clockwise disk scheduler also uses an EDF scheduling technique, but it uses a novel technique to calculate the

Symbol	Name	Meaning
τ_i	Task	*I*-th task of a task set
T_i	Period	Period of a task *i*
C_i	Service time	Service time of task *i*
	Request	Single invocation of a task
r	Release time	Time at which a request is queued
d	Deadline	Time at which a request must be finished
U	Utilization	Load on resource
L	Variable	Used to determine schedulability of a task set
ΔL	Slack time	Minimum time between end of request and deadline
lcm		Least common multiple
$W_i(t)$	Resource demand	Cumulative processor demand over period t for i tasks
$M(L)$	Slack time	Slack time over distance L
$N(i, L)$	Slack time	M(L) with an ordered request from task τ_i
E	Service time	Expected service time
M	Memory	Memory usage
D_j	Disks	Number of disks in a dynamic partition for client j
b_j	Size	Buffer size used by client j

Table 6.1: Terminology

slack in the schedule in advance. This slack time, called ΔL, is the minimum time between the completion of any real-time request and its deadline. Since this minimum slack time is available *after* the execution of each real-time request, it can also be applied *before* each real-time request. The ΔL scheduler only executes best-effort requests before a real-time request if there is ΔL slack time available. To do this, the scheduler first determines the predicted service time of a candidate best-effort request. Next, it decides between a best-effort and a real-time request by comparing the predicted best-effort request's service time to the available ΔL slack time. When compared to the aforementioned LST scheduler from Symphony, the ΔL does not need to traverse the entire list of queued requests to find slack time. Instead, it only needs to maintain a parameter that reflects the available remaining slack time.

Real-time scheduling systems make use of a schedulability test to guarantee that a task set is schedulable. Such schedulability tests guarantee that even in the most critical case, all requests that are invoked by tasks can be scheduled and none of these requests miss a deadline. Clockwise ΔL scheduling is based on a proper schedulability test for nonpreemptive dynamically-scheduled resources. In contrast to, for example, CPU scheduling, disks cannot be scheduled preemptively. This means that a schedulability test needs to guarantee that there is sufficient time in the schedule to finish *any* request that cannot be preempted while also guaranteeing that *any other* request still meets its deadline. The schedulability test is based on work by Jeffay *et al.* [53] and is explained later in Section 6.1.2.

Table 6.1 lists a number of terms that are used throughout this chapter.

The remainder of this chapter is organized as follows. Section 6.1 gives an overview of existing real-time scheduling techniques and its applicability to schedule disks. Section 6.2 describes

an extension to Jeffay *et al.*'s nonpreemptive EDF scheduling that allows real-time requests to be delayed by the precomputed ΔL slack time without missing a deadline. Section 6.3 describes the applicability of the new scheduling algorithm in Clockwise and it describes how to use schedule-slack time in a real system. Section 6.4 presents current approaches in scheduling a mixed-media load. Section 6.5 presents simulations with the ΔL scheduling technique.

6.1 Real-time scheduling techniques

There has been considerable attention to real-time scheduling problems – the problem of guaranteeing that invocations finish *on time*. Most of the systems that have been built and scheduling algorithms that have been published discuss issues of controlling time-critical devices: a catastrophic system error occurs when a task invocation misses a *deadline*. These systems are called hard real-time systems. Most published articles use the example of controlling the motion of a robot or controlling a nuclear power plant.

To guarantee that applications that run in a hard real-time system meet their deadlines, the worst-case service times of task invocations are known a priori. A hard real-time system's *schedulability test* makes sure that in all cases the entire task set can run without missing any of the deadlines when all of the task invocations use at most the worst-case service time.

Soft real-time systems use the same techniques as hard real-time systems, but have relaxed the schedulability tests and/or they use average invocation service times rather than worst-case service times. Instead of guaranteeing that all deadlines are met, a soft real-time system often offers only statistical guarantees: there is a chance that a deadline is not met, even though the task is admitted by the system. When such a task invocation misses a deadline, no real harm is done. Continuous-media scheduling is often presented as an example.

The way soft real-time systems handle transient overload is of particular importance: when request[1] service times are underestimated there is a chance that the system becomes overloaded. Most soft real-time systems deal with this situation by reducing the quality of the offered service. If, for example, a CPU is scheduled and an overload occurs, the number of CPU cycles given to task invocations is usually reduced. It is common practice to reduce the quality of the task that caused the overload.

Each task invocation in a real-time system has several characteristics: a *release time*, *deadline*, *service time* and a *priority*. The release time is the earliest possible time at which an invocation can start. The deadline of a task invocation is the time at which the invocation must be finished. The service time of an invocation is the time the invocation requires to execute. Task invocations are ordered in a real-time system by their *priority*: usually the highest-priority task invocation is executed first. Consider for example, the scheduling of invocations from two tasks τ_1 and τ_2. As is shown in Figure 6.1 each invocation has a release time and deadline and requires a service time to execute.

Much effort has gone into preemptive (CPU) scheduling: a high-priority task invocation can interrupt and preempt a low-priority task invocation in order to meet its deadline. If, for example,

[1] The terms invocation and request are used intermittently.

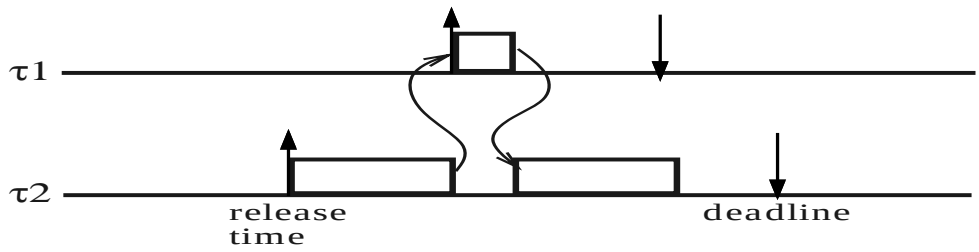

Figure 6.1: Preemptive real-time scheduling.

a low-priority invocation is executing on a CPU and a high-priority task invocation enters the system, it is given the CPU thereby preempting the low-priority invocation as is shown in Figure 6.1. The scheduler's schedulability test guarantees that even when low-priority task invocations are preempted, all task invocations still meet their deadline. The preemption time between two task invocations is usually assumed to be zero.

Although preemptive real-time scheduling is an important field and certainly has applicability to various real-time problems, it is not directly applicable for disk scheduling. Given current disk controllers, it is usually not possible to preempt an earlier started low-priority disk request, and even when this can be done, the preemption overhead can be quite high. A preempted disk misses at least a rotation per preemption, which costs between 6 and 9 ms for modern disks and it is likely that two seeks are required between the low-priority request's track and high-priority request's disk track. Roughly, one can say that at least 10–20 ms are wasted for performing a single preemption. Transferring a megabyte block from a Seagate Cheetah takes approximately 75 ms, so the context-switch time cannot be assumed zero.

Instead of scheduling disks by a preemptive scheduler, they are better scheduled by a *non-preemptive resource scheduler*.

6.1.1 Liu and Layland

Early and important work on real-time scheduling techniques is the work by Liu and Layland [66] on *Rate-Monotonic* (RM) and *Earliest Deadline First* (EDF) scheduling. They have proposed two scheduling algorithms to schedule tasks in a hard real-time environment. Their algorithms are based on fixed- and dynamic-priority scheduling. The fixed-priority (RM) scheduler determines the priority of a task before the task is scheduled (*i.e.*, off-line), the deadline-dynamic (EDF) scheduler determines the priority of a task invocation during a run (*i.e.*, online). Liu and Layland assume that task invocations can be preempted.

The RM scheduler assigns priorities in such a way that the task that uses most resources is given the highest priority. It is proven that such a priority assignment leads to an optimal scheduler for fixed-priority task sets: if a task set can be scheduled with any fixed-priority assignment, then the RM scheduler can also schedule the task set.

Liu and Layland show that the *Least Upper Bound* (LUB) utilization of the RM scheduler is:

$$U_{lub} = n(2^{\frac{1}{n}} - 1)$$

Here n represents the number of tasks in the task set. The U_{lub} means that if the utilization U:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

with C_i representing the service time that is required for an invocation of task i and T_i representing task i's period, is below the LUB of the RM scheduler, the task set is feasible (*i.e.*, no task invocations miss their deadline). For large n, $U_{lub} \simeq \ln 2$ (≈ 0.69).

The U_{lub} of the RM priority assignment does not imply that a specific task set can not achieve higher utilizations. An example is when task periods are congruent: the utilization of the processor for a task set in that case can be as high as 1.

Liu and Layland also introduced a *critical instant* for a fixed-priority scheduler. A critical instant is when a low priority task is released simultaneously with all tasks of higher priority: the low priority task needs to wait for all other higher-priority tasks to complete. It is proven that the worst-case release times for a periodic task set is when all tasks are released simultaneously, or all release times are 0.

In EDF scheduling,[2] the task invocation with the earliest deadline is given the highest priority and can preempt a task invocation that has a later deadline (*i.e.* lower priority). It is proven that iff:

$$\sum_{j=1}^{n} \frac{C_j}{T_j} \leq 1$$

a task set is *feasible*. Also, the EDF scheduling algorithm is optimal: if a task set can be scheduled by any dynamic algorithm, it can also be scheduled by an EDF scheduler.

6.1.2 Nonpreemptive (EDF) scheduling

Jeffay *et al.* [53] have determined the necessary and sufficient set of conditions for a set of sporadic or periodic tasks with arbitrary release times to be schedulable by a nonpreemptive version of Liu and Layland's EDF scheduler. Assume a task set $\tau_1 \ldots \tau_n$ is sorted in non-decreasing order by period. If the task set is schedulable then the following two conditions must hold:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{6.1}$$

$$\forall i, 1 < i \leq n; \forall L, T_1 < L < T_i; L \geq C_i + \sum_{j=1}^{i-1} \lfloor \frac{L-1}{T_j} \rfloor C_j \tag{6.2}$$

Condition (6.1) says that the total load may not exceed 1 and is identical to Liu and Layland's EDF admission test. Condition (6.2) says that the processor demand in period L must always be less than or equal to the length of the interval such that any low priority task invocation that is started at instance $t = -1$ can run with all higher priority task invocations that are released at $t = 0$ without preempting the low priority task invocation as is shown in Figure 6.2.

[2]This type of scheduling is also called deadline-dynamic scheduling.

Jeffay *et al.* also proof that a nonpreemptive version of Liu and Layland's EDF scheduler is universal for periodic and sporadic tasks. This means that when any nonpreemptive task scheduler schedules a task set, so does nonpreemptive EDF.

When initial release times are associated with a task set, the task set becomes a concrete task set. The authors first proof that if a sporadic task set is schedulable, then any *concrete* sporadic task set is schedulable and therefore also schedulable by a nonpreemptive EDF scheduler. The reason for this is shown by the authors: the schedulability of concrete sporadic task set does not depend on the actual release times of task invocations.

Determining if task sets that are rejected by (6.1) and/or (6.2) are schedulable when they are made concrete is shown to be a NP-hard problem in the strong sense. When a periodic task set is not schedulable, it can generate concrete task sets that are schedulable and concrete task sets that are not schedulable. Unless P = NP, a pseudo-polynomial time algorithm does not exist for deciding if a concrete periodic-task set is schedulable. The authors present a polynomial-time transformation from the 3-PARTITION NP-hard problem to the schedulability analysis of a concrete periodic task set.

Korst *et al.* [57] have determined the necessary and sufficient conditions to schedule two tasks nonpreemptively on a single processor with arbitrary release times. Also, a method is introduced to combine two tasks into a larger task and an algorithm is described to assign tasks to parallel processors (*e.g.*, multiple disks).

6.1.3 Real-time servers

There has been considerable attention to the concept of *servers* in real-time work. A server is a periodic task whose purpose is to service aperiodic requests as soon as possible. A polling server, for example, runs periodically and executes pending requests from aperiodic tasks [16, 61]. A polling server is usually assigned the remaining processing capacity on a resource and it is scheduled accordingly. Aperiodic invocations need to wait for the next *poll* to be executed. Permutations of the polling server are the sporadic- and deferrable-server algorithms. A background server is the most simplistic version of the servers: it only executes when there is no real-time work queued.

Lehoczky *et al.* [16, 61] introduced a slack time stealing algorithm to schedule aperiodic tasks in the slack time of a periodic schedule in a fixed-priority preemptive environment. The proposed algorithm is proven to be optimal in the sense that it minimizes the response time in every aperiodic task compared to all other scheduling algorithms. The basic idea behind this scheduler is to find and predict all 'slack' time in the schedule so that the periodic task invocations do not miss their deadlines. The optimality proof shows that this algorithm maximizes the use of slack periods by aperiodic tasks. It is also shown that the aperiodic response times yielded by the slack time stealer algorithm are close to what can be predicted by using queuing models. Also, the aperiodic tasks can be scheduled in a hard real-time manner when the service time of the aperiodic task is less than or equal to the amount of slack in the schedule.

The slack time stealing algorithm from Lehoczky *et al.*, which is implemented as a real-time *server*, is shown to be superior to other aperiodic task servers. In particular, it leads to faster response times compared to the background server and the polling server. In a sense, the slack

time stealing server is not a true real-time server since there is no periodic task for the server. The slack time stealing algorithm tries to run periodic real-time invocations as late as possible to give way for aperiodic invocations.

Clockwise's ΔL disk scheduler is a scheduling approach that is equivalent to Lehoczky's slack time stealing server algorithm except that Clockwise's approach work for a nonpreemptively scheduled resource. Lehoczky *et al.*'s approach is not applicable for disk scheduling because their slack time stealing server is one that is designed for a fixed-priority preemptive scheduler.

Several dynamic priority real-time servers exist, such as a dynamic priority exchange server, dynamic sporadic server, *Total Bandwidth Server* (TBS) and an *Earliest Deadline Late* (EDL) server [17]. The dynamic priority exchange server exchanges run time with that of lower priority tasks when there are no aperiodic invocations to be executed. When aperiodic invocations arrive in the system, the service time can be reclaimed from the system. The dynamic sporadic server replenishes the capacity of the server only when aperiodic invocations have consumed resources.

The total bandwidth server schedules aperiodic invocations by considering the utilization. In this server, the release time and deadline of an invocation are determined through an invocation's last deadline, the requested service time and the utilization as follows:

$$d_k = max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

The deadline of invocation k is d_k, the release time of invocation k is r_k, C_k represents the service time for invocation k and U_s is the TBS server's utilization. With this server, aperiodic invocations are given the earliest possible release times and deadlines in an EDF schedule.

The EDL server can be considered a dynamic version of Lehoczky's slack time stealing algorithm for a preemptive deadline-dynamic scheduler. In EDL, the slack periods are found by moving the periodic invocations in a hyper-period as close as is possible to their deadlines. A hyper-period is defined as $H = lcm(T_1, \ldots, T_n)$. Aperiodic tasks are invoked in the slack periods, which are found in the hyper-period schedule. Clockwise's ΔL scheduler can be considered an EDL server for best-effort traffic in a nonpreemptive EDF environment. Fortunately, Clockwise does not need to calculate the schedule in a hyper-period.

6.1.4 Other real-time work

Sha *et al.* [92] describe some practical problems with RM and deadline-dynamic priority scheduling. The authors describe the stability problems of both schedulers. It is not uncommon that priority assignments and schedulability tests are performed on the average execution times rather than the worst-case execution times. This means that it is possible that a *transient* overload occurs. When task invocations are assigned priorities through a deadline-dynamic scheduler, it is unpredictable which task invocations miss deadlines. When tasks are scheduled by a RM scheduler, the authors argue that the tasks with the longest periods miss deadlines.

Sha *et al.* also describe a *Time Division Multiplexer* (TDM) to solve the stability problem. Time is subdivided into a number of slots and tasks are assigned to slots. A dispatcher periodically examines all of the slots in a cyclic manner. There are different major and minor cycles

through the slots. Major cycles are given the highest priorities and critical tasks are assigned to these major cycles. The dispatcher guarantees service of the major cycles. The problem with TDM, however, is that it is a time consuming task to assign and modify the tasks to the cycles. Usually all of the tasks are hand-crafted into the slots. Most continuous-media systems use a TDM disk scheduling algorithm.

Sha *et al.* propose a period transformation method where mission critical tasks are given the shortest periods. This means that under the RM scheduling regime, mission critical operations are given the highest priorities. Since short-period tasks are not the tasks that miss a deadline in case there is a transient overload, mission critical tasks have a higher chance to meet their deadlines. Two transformation methods are proposed that extend or reduce the periods of tasks.

Finally, Sha *et al.* argue that an integrated scheduler for hardware and software processes improves the utilization of machines. Simulations show that the combination of FIFO ordering in many hardware policies and RM or deadline-dynamic driven schedulers is a bad one: the performance quickly deteriorates.

Lehoczky [62] proves that a task τ_i can be scheduled for all task invocation release times using the RM scheduling algorithm iff:

$$L_i = \min_{0<t\leq T_i} L_i(t) \leq 1$$

with

$$L_i(t) = \frac{W_i(t)}{t}$$

and

$$W_i(t) = \sum_{j=1}^{i} C_j \lceil \frac{t}{T_j} \rceil$$

and the entire task set $\tau_1 \ldots \tau_n$ can be scheduled iff:

$$L = \max_{1\leq i\leq n} L_i \leq 1$$

Here C_i and T_i are the service times and periodicity of task τ_i. $W_i(t)$ gives the cumulative demands on the processor made by the tasks over $[0, t]$ when 0 is a critical instant. $L_i(t)$ is a piecewise monotonically decreasing function with discontinuities at the RM scheduling points, *i.e.* points where new tasks enter the system. To verify that a task set is schedulable, the Liu and Layland U_{lub} is used on a task set and when the load is more than this bound, it suffices to test whether $L \leq 1$.

Lehoczky [60] analyze schedulability tests for task sets with arbitrary deadlines in a preemptive environment. In this work a deadline for invocation i of task j is not defined by the release time of run $i + 1$. Instead, a Δ is introduced that represents a common deadline postponement factor and all deadlines in a task set are extended with this factor Δ. It is shown that by using a Δ of 2, the U_{lub} on the utilization increases from $\ln 2$ to 0.811 and when a Δ of 3 is used, the U_{lub} becomes 0.863. The lesson from this work is that by extending the deadlines, the overall utilization increases substantially. Also note that when $\Delta = 1$, *i.e.* the original deadline is used, the U_{lub} equals $\ln 2$.

Harbour *et al.* [32] consider the problem of fixed-priority scheduling with tasks that can have a varying priority during the execution of an invocation. This is in particular important if the task invocation can be activated from, for example, a hardware interrupt service routine. The authors present a framework in which one can reason about time given to such tasks and a method of analysis of the tasks.

Leung and Whitebread [68] describe deadline-monotonic scheduling systems, which are systems where the deadlines for task invocations are not equal to the period of a task. Audsley *et al.* [5] present two sufficient but not necessary (*i.e.*, over-conservative) schedulability tests and a complex sufficient and necessary schedulability test for these deadline-monotonic task sets. This class of schedulers enable the scheduling of 'dead time' between the release time and deadline of an invocation and enable the modeling of sporadic task sets. The first sufficient but not necessary schedulability test is shown to be overly pessimistic. This test is based on the concept of *critical instants* and calculates the maximum influence of higher-priority task invocations on the analyzed task invocation. A slightly more complicated analysis takes into account that not all higher-priority task invocations that are released before the deadline of the analyzed task invocation, consume processor resources. Only by knowing the exact interleaving of task invocations a necessary and sufficient schedulability test can be constructed. Finally, authors model sporadic processes onto deadline monotonic task sets.

Katcher *et al.* [54] present simple and straightforward schedulability tests that include operating-system overheads. In particular, the time to process an interrupt, to run the scheduler, to store and retrieve the process' context and to discard of the process are also taken into consideration. The authors show through simulations that when these values are also taken into account, the utilization can be substantially lower than what is predicted by assuming these values are zero, which is the usual practice in most real-time work. A system simulator is constructed that uses real values taken from a Sony News Workstation that runs Real-Time Mach [99].

Priority-inheritance work, priority-ceiling protocols and real-time transactions [51] all study ways to deal with locked resources in a preemptive real-time environment. Given that all these schedulers assume that task invocations are preemptive, these techniques cannot be used directly to schedule disks. Only in the case that a single shared resource is modeled in such schedulers and the resources can be locked throughout the entire execution of invocations, these scheduling techniques can be applicable for scheduling disks. Further research is required to investigate this.

There exist a large number of other real-time scheduling approaches. Pin-wheel scheduling [30] tries to minimize jitter in a schedule also in a preemptive setting. Manabe *et al.* [69] present a new feasibility test for rate- and deadline-monotonic scheduling. However, all of these approaches have little applicability to Clockwise's disk scheduling and are therefore omitted.

6.2 Clockwise disk scheduling

Clockwise scheduling is based on work by Jeffay *et al.* [53]: real-time disk tasks are scheduled by a nonpreemptive EDF scheduler. Jeffay's schedulability test uses task periods and task request service times to determine if a task set is schedulable by a nonpreemptive EDF scheduler. In Clockwise, these service times and periods are defined by the storage application's QoS settings.

The challenge for the Clockwise disk scheduler is to mix best-effort requests into the stream of real-time requests such that, on one hand real-time requests do not miss deadlines and, on the other hand best-effort requests are serviced quickly. When a best-effort request must wait behind a queue of (bulky) real-time requests, the latency of the best-effort request can be severe. So, to schedule both types of traffic on the same disk, it is advantageous to execute the best-effort request before the real-time queue of requests.

Requests from tasks that are not considered by the schedulability test to determine their schedulability are called *unadmitted* requests. Currently, there is no fundamental knowledge of the effects of executing unadmitted best-effort requests on a nonpreemptively (EDF-ordered) scheduled task set. Also, there is no knowledge how to prioritize unadmitted best-effort requests in a nonpreemptively (EDF-ordered) scheduled task set. Jeffay *et al.* only dictate that task requests need to be executed in strict EDF order without considering unadmitted best-effort load.

Intuitively, it seems that by extending the deadlines of best-effort requests to infinity, these unadmitted requests can be mixed into the stream of real-time requests. Best-effort requests are, in this case, only executed when there are no more real-time requests to execute. By setting the best-effort deadlines to infinity such tasks do not contribute to the load by definition of (6.1) and the load is never higher than it is without the best-effort load. However, as is shown later in the section, condition (6.2) is vital to understanding why this approach does not work: executing a best-effort request when there are no real-time requests available may lead to deadline misses of real-time requests because of the nonpreemptiveness of disks.

Another intuitively appealing approach is the approach of Symphony [93]. Symphony calculates a *Latest Start Time* (LST) schedule for the real-time load, and when there is sufficient time to execute a best-effort request before the latest start time of a real-time request, the best-effort request is executed *before* the real-time request. However, this approach fails for the same reason as is described above: the scheduler does not account for the nonpreemptiveness of the best-effort request and real-time requests may miss a deadline.

To understand how to schedule unadmitted best-effort requests in a non-preemptive EDF schedule, it is important to consider (6.2). Figure 6.2 presents four tasks that are scheduled by a nonpreemptive EDF schedule. The key to understanding (6.2) is analyzing the schedule that is presented in the figure. The four tasks in the figure are all released twice and the second release time equals the deadline of the first invocation. Every task $\tau_j, 1 \leq j \leq 4$ has a period T_j and every task request has a service time C_j. At time $t = -1$, only a request from task τ_4 is released. Since there are no other requests released, τ_4 is started immediately and finishes at $C_4 - 1$. At $t = 0$ all of the other tasks release a request. Condition (6.2) guarantees that any distance L between T_1 and T_i is long enough to execute both the unpreemtable request that started at $t = -1$ *and* any of the requests that can be released and have a higher priority. *I.e.*, it guarantees that requests from $\tau_1 \ldots \tau_3$ meet their deadlines even though a request from τ_4 is started at $t = -1$. Since (6.2) tests all possible combinations, the schedulability of the task set is established.

Clockwise disk scheduling uses (6.1) and (6.2) to determine the schedulability of a task set and it uses both conditions to calculate the minimum slack time that is hidden in the schedule. Intuitively, the right hand side of (6.2) iterates over all possible resource demands including all possible interferences that can occur for any distance L. The difference between distance L and all possible resource demands is the *slack time* between the requests and their deadlines. The

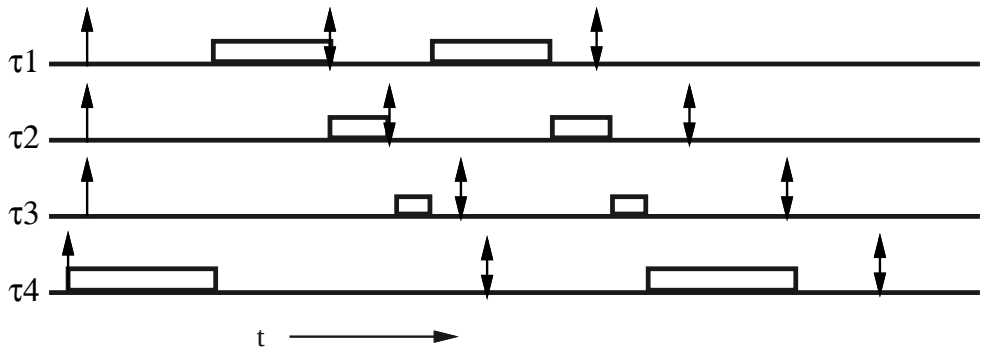

Figure 6.2: Non-preemptively scheduled tasks.

minimum represents the minimum time between any combination of requests and the deadlines of requests. This slack time is called the ΔL slack time.

If any real-time request completes at least ΔL before its deadline, the real-time requests can also start at most ΔL later in the schedule without missing a deadline. It is quite possible in nonpreemptive EDF scheduling that delaying a real-time request by at most ΔL can result in later start times later in the schedule. However, since ΔL is the minimum time from any real-time request to its deadlines, delaying such a request by at most ΔL does not cause it to miss a deadline.

Clockwise uses ΔL to decide whether or not to give precedence to a best-effort request when real-time requests are waiting to be executed. If a real-time request can be delayed (*i.e.*, ΔL is not used up already), and the predicted service time of the best-effort request is less than the slack time, the best-effort request is started instead of the real-time request. Applying ΔL to the nonpreemptive EDF schedule, means that the task set is scheduled by some form of a just in time scheduler.

The ΔL scheduling technique corresponds to the slack time stealing algorithm by Lehoczky *et al.* [61]. The difference between the ΔL slack time stealing and Lehoczky's technique is that the ΔL slack time stealing works for a nonpreemptive EDF schedule. Lehoczky's slack time stealing technique only works in a preemptive scheduling environment.

The remainder of this section proofs that any real-time request finishes at least ΔL before its deadline when its task is admitted by the schedulability test and reasons how to apply the schedulability tests in a real system.

Definition 1. *Assume a task set $\tau_1 \ldots \tau_n$ is sorted in non-decreasing order by period. ΔL, the slack time of each real-time request, is defined by:*

$$
\begin{aligned}
&M(L) = L - \textstyle\sum_{j=1}^{n} \lfloor \frac{L}{T_j} \rfloor C_j \\
&\Delta L_m = \min_{T_1 \le L \le T_n} M \\
&N(i,L) = L - (C_i + \textstyle\sum_{j=1}^{i-1} \lfloor \frac{L-1}{T_j} \rfloor C_j) \\
&\Delta L_n = \min_{1<i\le n; T_1 \le L \le T_n} N \\
&\Delta L = min(\Delta L_m, \Delta L_n)
\end{aligned}
$$

Theorem 1. *If a task set is schedulable by a nonpreemptive* EDF *scheduler,* i.e. *it satisfies (6.1) and (6.2), then each request completes at least ΔL before its deadline.*

The proof to this theorem follows directly from Theorem (4.3) of Jeffay *et al.* It is established by deriving upper bounds to the load for any distance L.

There are two cases to consider:

To determine the slack time of a set of requests that are executed in deadline order, *i.e.*, no low priority request precedes a high-priority request,[3] the first case of the definition for ΔL is considered: ΔL_m. The maximum demand for a period L is when all tasks release a request simultaneously: $\sum_{j=1}^{n} \lfloor \frac{L}{T_j} \rfloor C_j$. The executed load at some instance of $L_i = T_i, T_1 \leq T_i \leq T_n$ consists of requests from tasks $\tau_1 \ldots \tau_i$. Since requests are executed in deadline order, the last request that is executed is a request from task τ_i. Hence, the slack time for the request from τ_i at L_i is given by $M(L_i)$. Other values for L_i do not have to be considered to determine the slack time for a request from τ_i: larger values for L_i lead to higher slack times and when smaller values for L_i are considered, requests from τ_i do not contribute to the load in $M(L_i)$. Since all possible values L are considered separately for $M(L)$, all requests from tasks $\tau_i, 1 \leq i \leq n$ are considered. Since ΔL_m is the minimum of all $M(L)$, ΔL_m is the minimum slack time when requests are executed in deadline order.

The second case is when a low-priority request precedes higher-priority requests. This case is considered separately by Jeffay *et al.* in case 2 of the proof to their Theorem (4.3) and is covered by the second part of the definition for ΔL: ΔL_n. The proof to Jeffay's Theorem (4.3) is based on deriving upper bounds to the load in any interval L given that the start time of the low-priority request precedes one or more high-priority requests by one instance. It is proven that the upper bounds to the load in distance L are a sufficient measure for the schedulability of the task set.

Assume a request from task $\tau_i, 1 < i \leq n$ is invoked at $t = -1$ and at $t = 0$ requests from all tasks with shorter periods than T_i are released. The request from task τ_i cannot be preempted. All requests from tasks $\tau_k, T_k < T_i$ that are released at $t = 0$ are invoked *after* the request from τ_i finishes. Hence, because of EDF scheduling, the minimum slack time after the execution of a request from task τ_k to its deadline is represented by $L_k - (C_i + \sum_{j=1}^{k} \lfloor \frac{L_k - 1}{T_j} \rfloor C_j)$ and $L_k = T_k + 1$, *i.e.*, $N(i, L_k)$. Since the tasks are scheduled through EDF, the request from task τ_k, having the largest period, is also the last request that is executed.

Any of the tasks τ_k that can be hindered by the servicing of a request from τ_i out of order has a period T_k that is at least one instance shorter than the period of T_i. All tasks τ_i except for task τ_1 are considered for out of order scheduling.[4] The minimum of $N(i, L_k)$ represents the minimum slack time after the invocation of a request from task τ_k when preceded by any other request from task $\tau_i, T_i < T_k$. Since ΔL_n iterates over *all* tasks i and distances L in $N(i, L)$, ΔL_n represents the minimum time to a request deadline when any other request is scheduled out of order.

Since ΔL is defined as the minimum of all possible slack times of the two cases, each task request finishes at least ΔL before its deadline. □

[3] Remember that under EDF, the task with the earliest deadline has the highest priority

[4] Requests from task τ_1 cannot be scheduled out of order.

The slack time ΔL is partly determined by considering requests that can be activated one instance before a higher priority task is released. This instance can be infinitely small, but is in practice bound by the resolution of the time variable. The minimum instance that can be represented in Clockwise is 1 ns.

To determine if a task set is schedulable, both (6.1) and (6.2) need to be evaluated. The first condition simply determines the utilization. However, (6.2) iterates over all possible values for L. If the task periods have a large deviation, or when the scale of L is small in comparison to the period, the number of iterations can be high. To reduce the number of calculations, Clockwise uses a technique that is also employed by Lehoczky *et al.* [62]. Rather than analyzing the entire range of L, Clockwise only examines the *scheduling points* of L. The next theorem establishes the validity of this approach.

Definition 2. *A scheduling point t_p is defined as the point in time where a new request is released.*

Theorem 2. *If a task set satisfies (6.2) in the scheduling points it also satisfies the condition throughout L.*

Proof: Define $W(j, L) = \lfloor (L-1)/T_j \rfloor C_j$, the total load for task τ_j for period L. Observe that $W(j, L)$ only changes when task τ_j reaches another scheduling point $L = n(T_j + 1), n = 1, 2, \ldots$ by value C_j. Next, define the sum $W_i(L)$ as the aggregate load of all tasks: $W_i(L) = \sum_{j=1}^{i-1} W(j, L)$. $W_i(L)$ only changes when one of the tasks $\tau_j, 1 \leq j \leq n$ reaches a scheduling point. This means that the right hand side of (6.2) remains constant between scheduling points, while L increases. So, if a task set satisfies (6.2) in the scheduling points, it also satisfies (6.2) between the scheduling points. □

Corollary 1. *If a task set satisfies (6.1) and (6.2) in the scheduling points, it can be scheduled by a nonpreemptive* EDF *scheduler.*

Proof: This follows directly from Theorem 2, Jeffay's Theorem 4.3 and Jeffay's Corollary 4.4. □

Given Corollary 1, condition (6.2) only needs to be analyzed in the scheduling points. Conditions (6.1) and (6.2) only need to be evaluated whenever a new task is admitted or when an old task is removed. Condition (6.1) executes in $\mathrm{O(n)}$ time complexity, where n represents the number of tasks in the task set. Condition (6.2) executes in $\mathrm{O(n^3)}$ time complexity where n also represents the number of tasks in the task set.[5]

6.3 Applicability

There are a number of problems that prohibit the direct use of the ΔL scheduling technique in Clockwise. In particular, to execute the schedulability tests, Clockwise needs to know of the periods and service times for each of the tasks. The Clockwise disk scheduler also needs to maintain a value that identifies the remaining slack time (ΔL_r) and it needs to know when the remaining slack time can be replenished with new slack time.

[5]The actual algorithm to determine the schedulability of a task set is only 77 lines of C-code.

6.3.1 Periods and service times

When a storage application requires real-time guarantees from Clockwise, it presents disk QoS parameters in the form of required bandwidth and block size that it uses for the transfers. These QoS parameters are used by Clockwise to determine the schedulability of a task set and to determine the internal disk scheduling parameters (such as *service times* and *periods*). Clockwise guarantees the storage application a minimum service that corresponds to the negotiated QoS contract. If the storage application requests more service from Clockwise than is negotiated for, the application is said to be *violating* its QoS contract: its real-time service degrades to a best-effort service.

The service times of requests (C_j) depend on the location, the layout of dynamic partitions and the user block size. To calculate the service time, Clockwise finds out which block addresses are used for the transfers and obtains the service times by using timing information from, for example, raw performance measurements (*e.g.*, such as the ones that are presented in Chapter 4). If a dynamic partition spans multiple disk zones, a transfer of data from or to a slow zone takes longer to complete than a transfer of data from or to a fast zone. Clockwise uses the worst-case disk service times from the slowest zone in the dynamic partition as a basis for the service-time prediction. Also, it adds the worst-case seek time for the disk.

Since the ΔL scheduler needs to know which of the blocks are used for a transfer, disk space needs to be preallocated on disk when real-time transfers are initiated. A Clockwise storage application either creates a dynamic partition beforehand and shrinks the size of the dynamic partition after it is written, or Clockwise allocates disk blocks on demand in such a way that all dynamic partition blocks that are allocated reside on the same disk zone. After a dynamic partition is created, it can be re-arranged to improve the schedulability of a task set. If, for example, a task set is unschedulable by the ΔL scheduler, an application can rearrange the layout of the dynamic partition such that the task set becomes schedulable. The schedulability of a dynamic partition can be improved by moving all blocks in the same disk zone, or by distributing the blocks to more parallel disks.

The period of a task is a function of the requested bandwidth, the block size that is used and the layout of a dynamic partition. Since dynamic partitions can span multiple disks, Clockwise first determines the per-disk fraction of data for the dynamic partition. This fraction is used to determine the per-disk bandwidth for a task. If, for example, a dynamic partition is stored on two disks, each disk needs to supply half the requested bandwidth (when seek overhead and rotational delay are ignored). Since the user block sizes are of fixed size, the disk periods are twice as long compared to a transfer of the same dynamic partition that is stored on a single disk. The disk's period for the application is determined by the time it takes for the storage application to fill or empty a block of data.

The disk period and service time are used for the schedulability test on a per disk basis. This means that a per-disk ΔL is maintained and implies that an application needs to maintain enough user buffers to keep all of the disks in the dynamic partitions busy. So, if an application uses n disks to store a dynamic partition, it requires $2n$ buffers for the transfer: one to queue in Clockwise, and one to use for its own purposes.

6.3.2 Using ΔL

When a real-time request arrives in Clockwise, Clockwise assigns a release time and deadline to the request based on the application's period T_i. The release time of a real-time request is determined through the deadline of the task's last request or the time at which the request is queued, whichever is later. If a task uses more resources than is anticipated for, the request is scheduled to be executed in the future, *i.e.*, it receives a future release time. A request's deadline is determined by adding the task's period to the calculated release time.

Clockwise maintains two queues per disk: one *First Come First Serve* (FCFS) best-effort queue and a real-time (EDF) queue. The best-effort queue holds all requests that are always executed in ΔL slack time, the EDF queue holds all real-time requests. The requests in the real-time queue are ordered by deadline. When real-time requests are scheduled to be executed in the future (*i.e.*, its task uses too many resources), the real-time service degrades to a best-effort service but the requests remain queued in the EDF queue. For as long as the release time has not yet arrived, the real-time requests are scheduled in ΔL slack time.

To schedule tasks in ΔL slack time, Clockwise maintains a value per disk that represents the remaining slack time, ΔL_r, for that disk. This parameter has a maximum value of ΔL and it is lowered every time a best-effort request is given precedence over real-time requests.[6] Clockwise only schedules best-effort requests before a real-time request when $\Delta L_r \geq E$, where E represents the expected execution time of the best-effort request. To determine E, Clockwise estimates the service times for the best-effort request by adding up the exact seek time from the current disk cylinder to the requested disk cylinder and the time to transfer the data from or to disk.

When ΔL_r is exhausted, real-time requests can no longer be postponed. Without further ado, requests from the real-time (EDF) queue are executed despite queued best-effort requests.

ΔL_r is replenished with new slack time when the real-time (EDF) queue is emptied or when all queued real-time requests are scheduled to be executed in the future. When all of the real-time requests have been executed, any newly released real-time request has, when finished, at least ΔL slack time to its deadline. This means that ΔL_r can be reset to ΔL. Note that during the period that no real-time requests are released, Clockwise can use this time to schedule requests without using time from ΔL_r. Since it is unknown when new real-time requests arrive, Clockwise can only schedule tasks with a maximum E of ΔL.

6.3.3 Best-effort request selection

The best-effort request selection policy determines which of the queued best-effort requests is scheduled in ΔL slack time. Clockwise maintains all best-effort requests in a best-effort FCFS queue and it maintains a list of real-time requests that are scheduled to be executed in the future in the EDF queue.

Requests in the EDF queue that are scheduled to be executed in the future (*i.e.*, their release times are set to the future), are said to be in violation because they send requests too quickly

[6] Also real-time requests that are scheduled to be executed in the future are considered since their service has degraded to a best-effort class of service.

to Clockwise compared to their negotiated QoS contract. Requests that are violating their QoS contract, have a violation rate: the difference between the contracted and used bandwidth. Since the EDF queue maintains all requests in Clockwise deadline order, requests that have a higher violation rate are queued later in the EDF queue than requests that have a lower violation rate.

When selecting violating real-time requests for execution in the ΔL_r period, the EDF queue is maintained. This means that tasks with higher violation rates are less likely to be executed in ΔL_r slack time. The reason for this type of scheduling is that it is only fair to give precedence to less violating requests.

Ordinary best-effort requests can be subdivided in synchronous and asynchronous request. The synchronous requests are requests on which a user waits for the results such as, for example, a directory or file read operation. Asynchronous requests are requests that only improve the system, *e.g.* a dirty file-block flush that is triggered by a periodic update timer.

To improve the perceived latency of best-effort requests, synchronous requests are given precedence over asynchronous requests. This means that when the disk scheduler selects a request, it looks for requests that are synchronous. To even further reduce the average latency of the requests, the disk scheduler schedules the best-effort requests in SCAN order [90, 50]. It finds all synchronous requests and orders them on cylinder number to minimize the effect of disk-seek operations.

There is, however, a limit to request reordering. Especially when data is written to disk, the order of the operations can be important when system failures are considered. Suppose there are two write operations W_a and W_b and b's contents depends on a's. In this case performing W_b before W_a may lead to faulty file-system states. When a UNIX Fast-File System, for example, creates a new file, it needs to write an inode for the new file, it needs to update the directory blocks in which the new file is named, and it needs to update the file-system's meta-data to reflect the allocation of a new inode and data blocks – all at different places at disk. These operations need to be performed in a particular order to accommodate for file-system repair applications such as *fsck(1)* after a system crash. A file-system repair application analyzes the state of a file system after a crash and based on its knowledge of the disk update order, it is able to determine when the system crashed and how to do or undo the last operation before the file-system crash.

Clockwise disk scheduling uses a technique that preserves the logical order on a per dynamic partition basis. Applications can tag operations to be part of a larger update, and when the disk scheduler selects a request with a tag, it makes sure that all of the preceding operations from the larger update have already completed.[7]

The disk scheduler needs to decide between the two classes of requests to schedule requests in ΔL time. Currently, Clockwise first executes all of the violating real-time requests before it schedules best-effort requests. This, however, implies that best-effort requests can *starve*. To prevent starvation, even best-effort storage applications can establish a QoS contract with Clockwise as is described later in Section 6.5.4. This means that all of the best-effort requests from a best-effort storage application are assigned release times and deadlines and that such best-effort requests are serviced with real-time service. It also means that there is a simple an effective

[7]The current Clockwise implementation only implements FCFS best-effort queue. Further experimentation is required to measure the effects of different ordering policies.

Task name	C_i	T_i
τ_1	42 ms	250 ms
τ_2	69 ms	250 ms
τ_3	122 ms	1 s
τ_4	163 ms	2 s

Table 6.2: A task set that is admitted by the USD scheduler and leads to deadline misses in reality.

way to decide which requests to run in ΔL_r time. All of the best-effort traffic is also ordered in Clockwise on deadline order and the disk scheduler picks the one with the earliest deadline.

6.4 Other approaches

Currently there are two other approaches with dynamically scheduled disks and these approaches are examined more closely: the *User-Safe Disk* (USD) [10, 9] from the University of Cambridge and the *Latest Start Time* (LST) scheduler from the Symphony system [93, 94]. Both schedulers schedule their requests using some form of EDF scheduler.

6.4.1 User-Safe Disk (USD)

USD attempts to divide the control and data path to disk such that all resources that are used for transferring data to or from disk are accounted to the application that caused the operation. The central USD server only executes the control functionality and divides the disk resource over the applications that make use of the disk. For an overview of USD, see Section 2.7.

USD uses the Nemesis CPU scheduler Atropos [83] to decide if a task set is schedulable. Atropos summarizes the load of each of the tasks and admits tasks if the load does not exceed one: $\sum_{j=1}^{n} C_j/T_j \leq 1$. This condition is identical to (6.1) and Liu and Layland's preemptive EDF scheduler's admission test. The condition is a sufficient test for a preemptive deadline-dynamic scheduler. If a high-priority request enters the system, *i.e.*, a request with a short deadline, under the preemptive EDF regime, a lower-priority request is preempted in favor of the high-priority request.

If, however, a request cannot be preempted, such as is the case for disks, the Atropos schedulability analysis admits task sets that in reality lead to deadline misses. An example of this is the task configuration as is shown in Table 6.2. For this configuration, four tasks are started on a single disk. The timings used are actual service times for reading or writing blocks on a Quantum Atlas-II disk. The service times are listed in column C_i, the periods are listed in the column T_i. The utilization of the task set listed here is 0.6475.

When the task set from Table 6.2 is scheduled by a nonpreemptive EDF scheduler, it immediately misses a deadline as is shown in Figure 6.3. At startup, requests from all tasks are released. The request from task τ_1 runs as the first request until $t = 42$ and is followed by requests from τ_2, τ_3 and τ_4. During the execution of τ_4, requests from τ_1 and τ_2 are released again. However, due to the nonpreemptiveness of the the request from τ_4, the request from τ_1 cannot start until

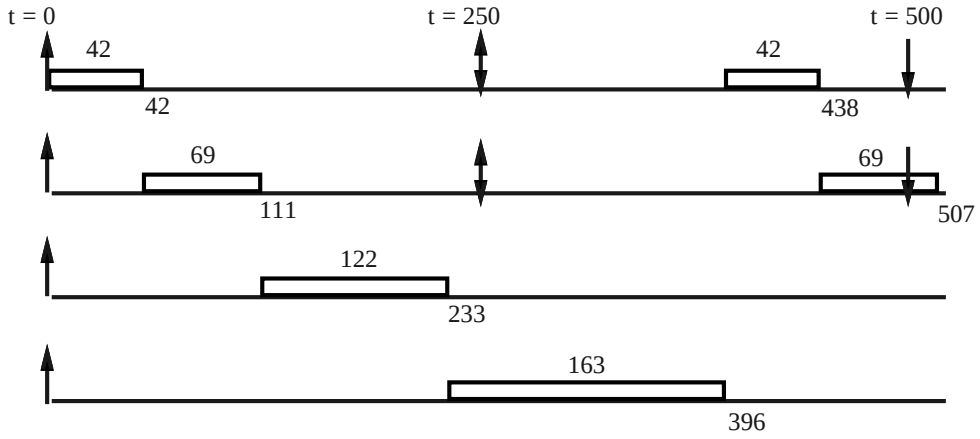

Figure 6.3: Non-preemptively scheduled tasks on USD.

$t = 396$. Following the request from τ_1, the request from τ_2 executes. The request from task τ_2 misses a deadline by 7 ms at $t = 507$. The reason for this is that the request from τ_4 cannot be preempted at $t = 250$ in favor of requests from τ_1 and τ_2 even though these requests have earlier deadlines.

When the Clockwise schedulability test is presented the task set that is listed in Table 6.2, the schedulability test rejects the entire task set. Clockwise comes to the correct conclusion that the task set is not schedulable by a nonpreemptive EDF scheduler.[8]

It is difficult to tell which task sets are admitted and which task sets are denied by Clockwise. As Jeffay *et al.* already pointed out: 'It is possible to conceive of both *schedulable* task sets that have a processor utilization of 1.0, and *unschedulable* task sets that have arbitrarily small processor utilization.' As soon as tasks need to be scheduled that have long computational times and that to run concurrently with short period tasks, the USD scheduler can be in trouble. USD tries to solve the deadline misses by reducing the service time of the task that missed the deadline [31]. As is already pointed out by Sha *et al.* [92] and as can be seen in Figure 6.3 such a policy reduces the QoS of a task that has not caused the overload situation in the first place.

6.4.2 Latest Start Time (LST) scheduling

The Symphony system uses a *Latest Start Time* (LST) scheduler that orders the real-time requests in EDF order. Whenever a request is released, the scheduler calculates the instance at which the request must start in order to meet its deadline. The latest start time of a request is calculated based on the service time of the request (*e.g.*: R_a) and the latest start time of the next request (*e.g.*: R_b) in the queue. If request R_b starts before the end of the deadline of R_a, the latest start time of R_a is set to the latest start time of R_b minus the service time of R_a. If this is not the case, the latest start time of R_a is set to the deadline of R_a minus the service time. Symphony resembles an EDL server, without calculating the hyper periods [17].

[8] In this respect, using ΔL scheduling in USD makes USD a User-Safer Disk.

Once the latest start time of queued real-time requests are known, Symphony can decide to schedule best-effort requests before real-time requests. The philosophy is that when the best-effort request completes before the latest start time of the first request in the real-time queue, the best-effort request can be executed prior the real-time requests without the real-time requests missing a deadline.

Assuming that Symphony uses Jeffay's schedulability analysis, the LST scheduler as is presented in [93] and [94] cannot guarantee that requests meet their deadlines. The reason for this is that the LST scheduler does not calculate the slack time ΔL: there is a chance that a real-time request misses a deadline if a best-effort request is started that uses more time than ΔL. To see this consider two tasks: τ_1 and τ_2. Requests from τ_1 have a service time of 10 and a period of 40 and requests from τ_2 have a service time of 20 and a period of 30. Also, consider that task τ_1 releases a request at $t = 0$. This request's deadline is set to $t = 40$ and the latest start time of the request is $t = 30$. At $t = 0$ a best-effort request arrives with a total service time of 25. Since the end time of the best-effort request is less than the latest start time of the request from τ_1, the best-effort request is executed. Next, at $t = 20$, a request from τ_2 is released. Its deadline is set to $t = 50$ and its latest start time is set to $t = 30$. Since the request from τ_2 now uses the service time that is meant for the request from τ_1, τ_1 requests' latest start time shifts to $t = 20$. Unfortunately, the best-effort request does not complete until $t = 25$ and the request from τ_2 misses a deadline at $t = 50$.

The problem with the LST scheduler is that a best-effort request may be given priority that violates the minimum-slack time ΔL. ΔL represents the *minimum* time between any completion of a real-time request and its deadline. When a best-effort request with a run-time longer than ΔL is started, it is possible that this request completes later than the latest start time of real-time requests that are released after the best-effort request has started. Only ΔL worth of best-effort requests may be scheduled in order not to miss real-time deadlines.

6.5 Simulations

For a true understanding of the value of the various schedulers in a mixed-media load, a number of simulations are performed on a disk simulator. A disk simulator is constructed that behaves much like a real Quantum Atlas-II disk and three schedulers are implemented and measured: a standard EDF scheduler, Symphony's LST scheduler and Clockwise's ΔL scheduler. For the experiments, the EDF scheduler works much like a background server [16]: only when there are no real-time requests queued, best-effort requests are executed.

The simulator simulates a disk I/O system that is constructed from three Quantum Atlas-II disks, and each one of them is connected by a private Fast SCSI-2 bus to a simulated host. The simulator configuration is similar to the configuration that is described in Chapter 4 and that is used for the raw I/O performance measurements. The simulator implements disk zones, rotational positions, head locations, and the disk's zero latency read policy. The disk's read performance of the simulator is first compared for a set of disk requests with true measured performance and the root square mean of the simulated performance is within 1 % of the measured

Name	Type	BW	Period (s)			
			256 KB	512 KB	1 MB	2 MB
WG	Motion-JPG	245 KB/s	0.886	1.772	3.543	7.087
RO	Motion-JPG	817 KB/s	0.313	0.627	1.253	2.507
RGB16	384×288	5.3 MB/s	0.047	0.094	0.189	0.377
RGB16	768×576	21.1 MB/s	0.012	0.024	0.047	0.094
CD	Uncompressed	172 KB/s	1.488	2.977	5.953	11.907

Table 6.3: Average bandwidth requirements. WG is the video track of 'Wallace and Grommit: A Close Shave', RO is the video track of 'The Hunt for Red October.' RGB16 is 16 bits per pixel, RGB video. 384×288 represents a quarter PAL quality video, 768×576 represents full PAL. The task periods are listed for various user block sizes.

performance for a limited number of block sizes.[9]

The real-time load that is used by the simulator is based on the video loads that are described in Chapter 3 and in Table 6.3. The table lists for a number of audio and video tracks the amount of resources required. The compressed movie tracks require, depending on the quality of the recording, some bandwidth between 245 KB/s and 817 KB/s, uncompressed video consumes between 5.3 MB/s and 21.1 MB/s depending on the frame size and number of bits per pixel and uncompressed audio bandwidth for a CD quality signal is only 172 KB/s.

The periods that are associated with each of the tasks depend on the block sizes that are used. Block sizes that are small lead to small periods and relatively high overheads for seeking and low values for ΔL. Block sizes that are too large lead to wasted memory space. Table 6.3 also lists for all types the task period depending on user block size.

The request service times depend on the block sizes that are used, the disks that are used and the location of dynamic partitions on disks. A megabyte transfer on the outer zone of a Quantum Atlas-II disk, for example, takes on average 116 ms, while on the inner zone this takes approximately 184 ms. Table 6.4 lists for a number of block sizes the worst-case service time per block per zone. To calculate the worst-case service times in the experiments, the 95-percentile or higher of the raw measured service times are used and the longest seek time is added to the read or write time.[10]

The best possible values for ΔL occur when the periods of all tasks are approximately the same. To achieve this in the simulations, all tasks use approximately the same periods. This means that CD streams and the 'Wallace and Grommit: A Close Shave' movie use 256 KB blocks, and the 'Hunt for Red October' movie use 1 MB buffers. The raw video streams are not used in the simulations: given that the minimum bandwidth for an uncompressed PAL stream is already 5.3 MB/s, only a limited number of jobs are schedulable on 3 parallel Quantum Atlas-II disks. In fact, when only a single full PAL stream is used for the experiments already 75 % of the available aggregate bandwidth is used up by such a stream. For the simulations, the real-time load consists

[9]Note that the read-ahead and write-behind policies are switched off during the real measurements and the simulator does not implement these policies either.

[10]The actual location of the data on disk can also be used to determine the worst-case seek time.

Zone	128 KB	256 KB	512 KB	1 MB
0	42ms	55ms	81ms	134ms
1	42ms	55ms	83ms	140ms
2	42ms	58ms	85ms	143ms
3	42ms	57ms	86ms	146ms
4	42ms	62ms	89ms	145ms
5	42ms	63ms	89ms	152ms
6	42ms	63ms	89ms	154ms
7	43ms	63ms	95ms	163ms
8	44ms	63ms	99ms	173ms
9	50ms	65ms	102ms	178ms
10	50ms	67ms	107ms	189ms
11	50ms	69ms	114ms	204ms

Table 6.4: Worst-case service times on a Quantum Atlas-II disk per zone.

Disk	Requests	Load per minute	
		Average	Peak
0	200796	139.4	2654
1	3973	2.8	1777
2	193539	134.4	2840
3	2054	1.4	994
4	431	0.3	144
5	122730	85.2	2598
6	249801	173.4	2412
7	27683	19.2	1349

Table 6.5: Load on June 1st, 1992.

of pairs of audio and video tracks, and for each run of the simulator, either a ‘Wallace and Grommit: A Close Shave’ stream or a ‘Hunt for Red October’ audio/video pair is added to the total load.

The best-effort load that is injected simultaneously with the real-time load consists of load generated from the HP trace files that are described in Chapter 3. From the set of trace files, only the CELLO trace files are used for the simulations. All trace files together describe approximately 30,000,000 disk requests including the location on disk, the absolute start time and the amount of data that is read or written. Instead of executing all 30,000,000 requests on the simulated disks, the busiest day throughout the trace is selected for play back (June 1st, 1992). Table 6.5 shows the number of requests per disk, the average load per disk per minute and the peak load per disk per minute on that day for the CELLO trace file. A total of 801,007 disk requests are used for simulations.

Each CELLO disk from the trace is implemented by a private and simulated dynamic partition that is laid out on all simulated Quantum Atlas-II disks. A separate thread of control inside the

simulator injects disk requests independently of all other CELLO disks from the traces at the correct (simulated) time into the disk simulator.

Both the real-time tasks and the best-effort tasks prepare disk requests and they execute the requests by passing the requests to a simulated disk scheduler. For every run the disk scheduler is started with an EDF, LST or ΔL scheduler. The only difference between the three disk schedulers is the way the requests are queued and de-queued. The disk schedulers pass the requests from the disk queue to the simulated disk for execution. On completion of the request, the simulated execution time is recorded in a statistics library that, at the end of the run, reports the latencies for the best-effort requests.

The best-effort latencies for three different continuous-media data layouts are measured. The simplest data allocation is when continuous-media dynamic partitions are stored on all disks. In this case, if dynamic partition block n is assigned to disk d, dynamic partition block $n + 1$ is assigned to disk $(d + 1) \bmod 3$. This experiment is called 'rotation'. All disks in the array receive the same real-time load.

To measure the influence of spatial locality in the 'rotation' experiment, an experiment is performed where data blocks are assigned to random locations on disk.

Lastly, an experiment is performed where continuous-media dynamic partitions are assigned to single disks. When data is assigned to all available disks, memory consumption is high. By limiting the number of parallel disks for continuous-media dynamic partitions to single disks, memory consumption is reduced. Again, the three schedulers are compared. The 'memory optimized' experiment is also performed with twice as large buffers to measure the influence of large ΔL on the schedulability of disks. This experiment is called 'memory optimized (large)' experiment.

6.5.1 Sequential layouts

Figure 6.4 presents values of ΔL as function of a real-time load for various configurations. Note that the maximum value of ΔL on the left edge of the figure corresponds to the minimum period of its tasks: in the 'memory-optimized' experiment ΔL is one second, because the service periods of all tasks are chosen to be roughly one second. In the 'rotation' curve, the period is three seconds, because a task visits each disk in one request out of three. For the 'memory-optimized (large)' experiment, ΔL is two seconds, because a double amount of data is read in each request.

The 'rotation' figure consists of two parts: one part that runs from 1 MB/s to approximately 10 MB/s and one part that runs from 10 MB/s to 18 MB/s. The first part of the curve the value of ΔL is determined by the second part of the ΔL function. The second part of the curve the value for ΔL is mainly determined by the first part of the function for ΔL. At the end of the curve, there is hardly any slack time available and best-effort load is scheduled poorly after this point. The maximum value for ΔL is approximately 3 seconds when using the 'rotation' assignment. Since continuous-media data is assigned to all three disks in parallel and since the periods of the tasks per dynamic partition block are approximately 1 second, each disk needs to service a request for a task approximately once every 3 seconds.

Figure 6.5 presents the best-effort latencies for the three schedulers for all disks. The hori-

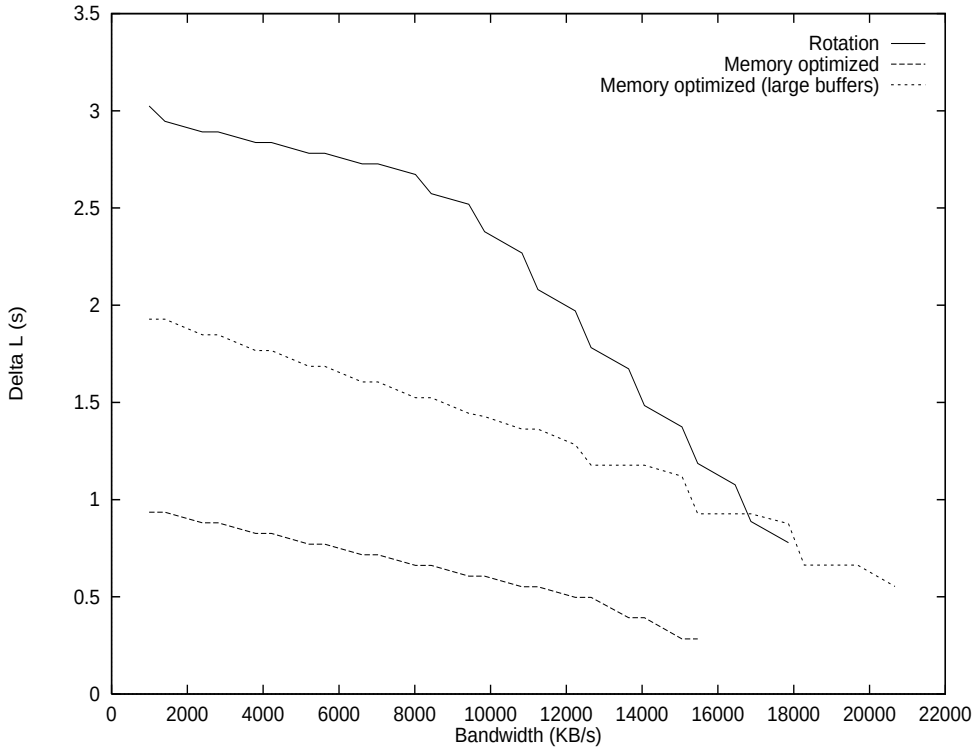

Figure 6.4: Aggregate real-time load and ΔL.

zontal axis presents the aggregate real-time bandwidth and the vertical axis presents the average best-effort latencies for all three schedulers (note that the best-effort latencies are presented on a logarithmic scale). The figures show that not giving priority to best-effort requests results in high latencies for such requests; the standard EDF scheduler schedules best-effort requests when there are no real-time requests queued and the best-effort latencies increase quickly. Clearly, this is undesirable. Both ΔL and LST scheduler perform much better up to medium real-time loads.

For low disk loads the LST and ΔL scheduler perform equally well. For both schedulers, the slack time in the schedule is sufficient to run all of the best-effort traffic before the real-time traffic. However, at some point the LST scheduler starts performing better than the ΔL scheduler.

The reason why the LST scheduler is better in scheduling best-effort traffic beyond a moderate real-time load is because it finds more slack time in the real-time schedule. The ΔL slack time is a minimum slack time: in any case each request has a slack of ΔL. The LST scheduler on the other hand calculates the actual latest start time of a request and finds more slack time compared to the ΔL scheduler.

For extremely high real-time loads, the LST scheduler performs in most cases (much) worse than the ΔL scheduler even though the LST scheduler finds more slack time. This is due to more *context switches* (*i.e.*, seeks) between real-time and best-effort dynamic partitions in the LST scheduler. In the simulation the real-time data blocks are grouped together and all best-effort CELLO disk data is grouped together. Switching from a best-effort area to a real-time area involves a (possibly long) seek of the disk arm.

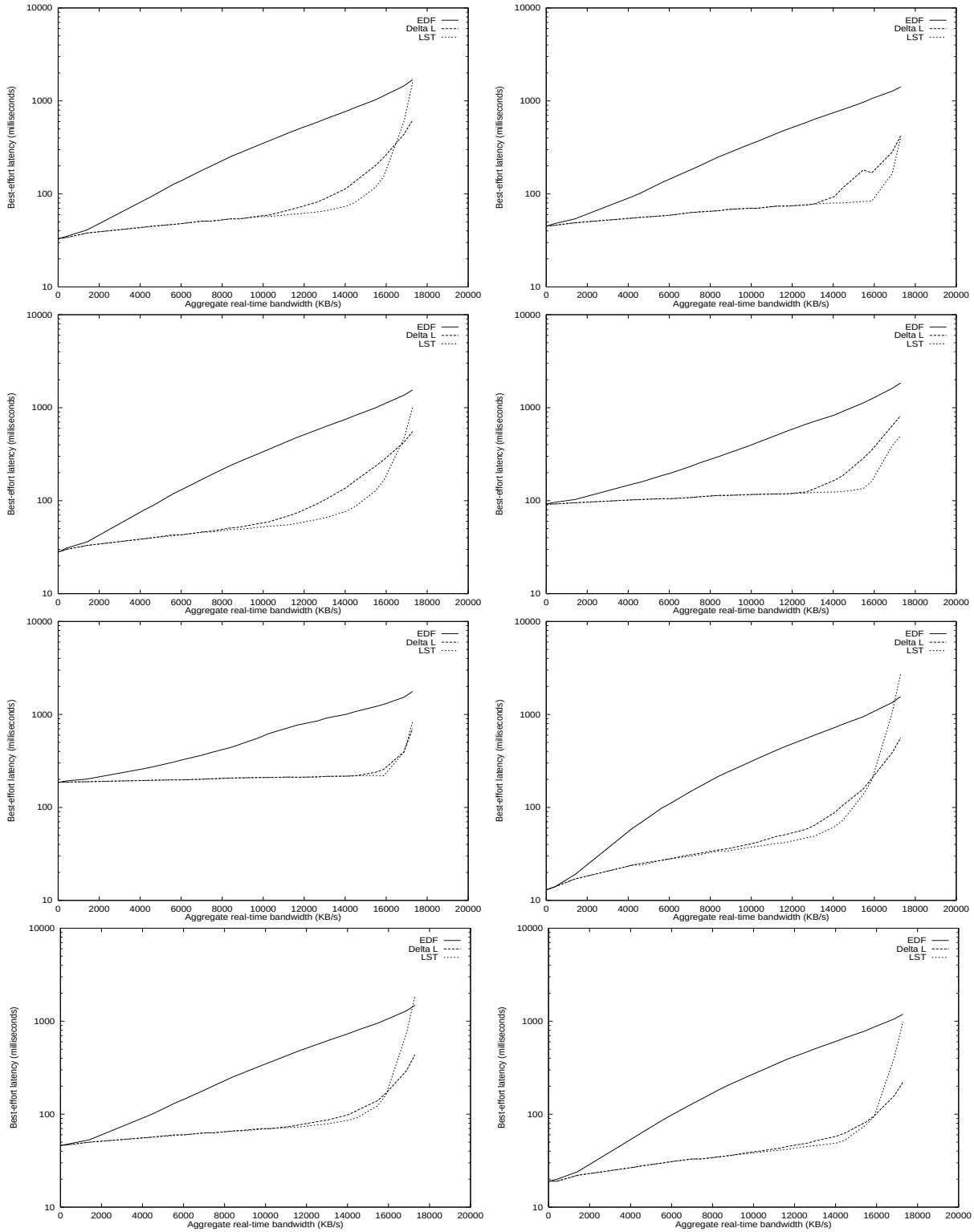

Figure 6.5: Best-effort latencies per HP traced disk ('rotation'). Upper left is CELLO disk 0, bottom right is CELLO disk 7.

Generally, a burst of real-time requests occurs in the LST scheduler when two or more requests have the same deadline, or when the execution times of the requests overlap. A burst of real-time requests occurs in the ΔL scheduler when ΔL_r is exhausted: all real-time requests are coalesced and best-effort requests are only scheduled again when the real-time queue is emptied. At that point ΔL_r is replenished.

In the experiments, best-effort requests suffer from shorter real-time request bursts when compared to the ΔL scheduler. This is because all of the real-time requests are moved to their deadlines in LST scheduling by using the worst-case timings. The actual service time of a request is usually shorter than the worst-case service time and a short quantum of time is left over after the real-time request completes, which is just enough to execute a few best-effort requests. This involves a (possibly) long seek, thereby wasting slack time. The ΔL scheduler executes a large number of best-effort requests in a single burst and the time to perform the seek operations is divided over a large number of best-effort requests.

Figure 6.6 shows the number of context switches between best-effort and real-time dynamic partitions (top figure). The figure shows that the number of context switches for both the ΔL scheduler and the LST scheduler increases when the number of real-time tasks increases. At approximately 35 real-time tasks the number of context switches for the ΔL scheduler and the LST scheduler start to differ. This is because at that point the ΔL scheduler decides at an earlier instance to start scheduling real-time requests instead of queued best-effort requests. The LST scheduler on the other hand, still schedules best-effort traffic, but since the found slack periods are small, only a limited number of requests are executed in the slack time. This means that the found slack time is more fragmented for the LST scheduler.

Figure 6.6 also shows that the number of context switches for the EDF scheduler reduces as the number of real-time jobs increases. This is because the real-time requests are always given priority over best-effort requests. When there are no more real-time requests queued, the EDF scheduler considers best-effort requests for execution. However, during the execution of the real-time requests, best-effort requests still arrive at the disk. This implies that the number of best-effort requests that is executed in a burst is larger than is the case for the other two schedulers.

The middle figure of Figure 6.6 shows the lengths of the best-effort bursts for all schedulers. The figure shows that beyond four real-time tasks, the number of requests that are executed in a burst by the EDF scheduler increases with the number of real-time tasks. The other two schedulers, on the other hand, reduce the best-effort burst length. Furthermore, the best-effort burst lengths in the other two schedulers are of equal length, which should imply that there is no difference in the number of context switches as is shown in the top figure. If, however, the standard deviation of the burst lengths is considered, as is presented in the bottom figure of Figure 6.6, it is shown that the ΔL scheduler has a larger deviation to the mean. This implies that the ΔL scheduler sometimes executes longer and sometimes shorter bursts. Since the number of context switches for the ΔL scheduler is less compared to the LST scheduler, it seems that the ΔL scheduler uses relatively long best-effort bursts during periods of high best-effort load.

The simulation measures the case that all real-time requests from the same task are released at exactly the same time. In reality these release times are probably not the same, thereby worsening the situation for the LST scheduler. The ΔL does not suffer from this effect because it coalesces

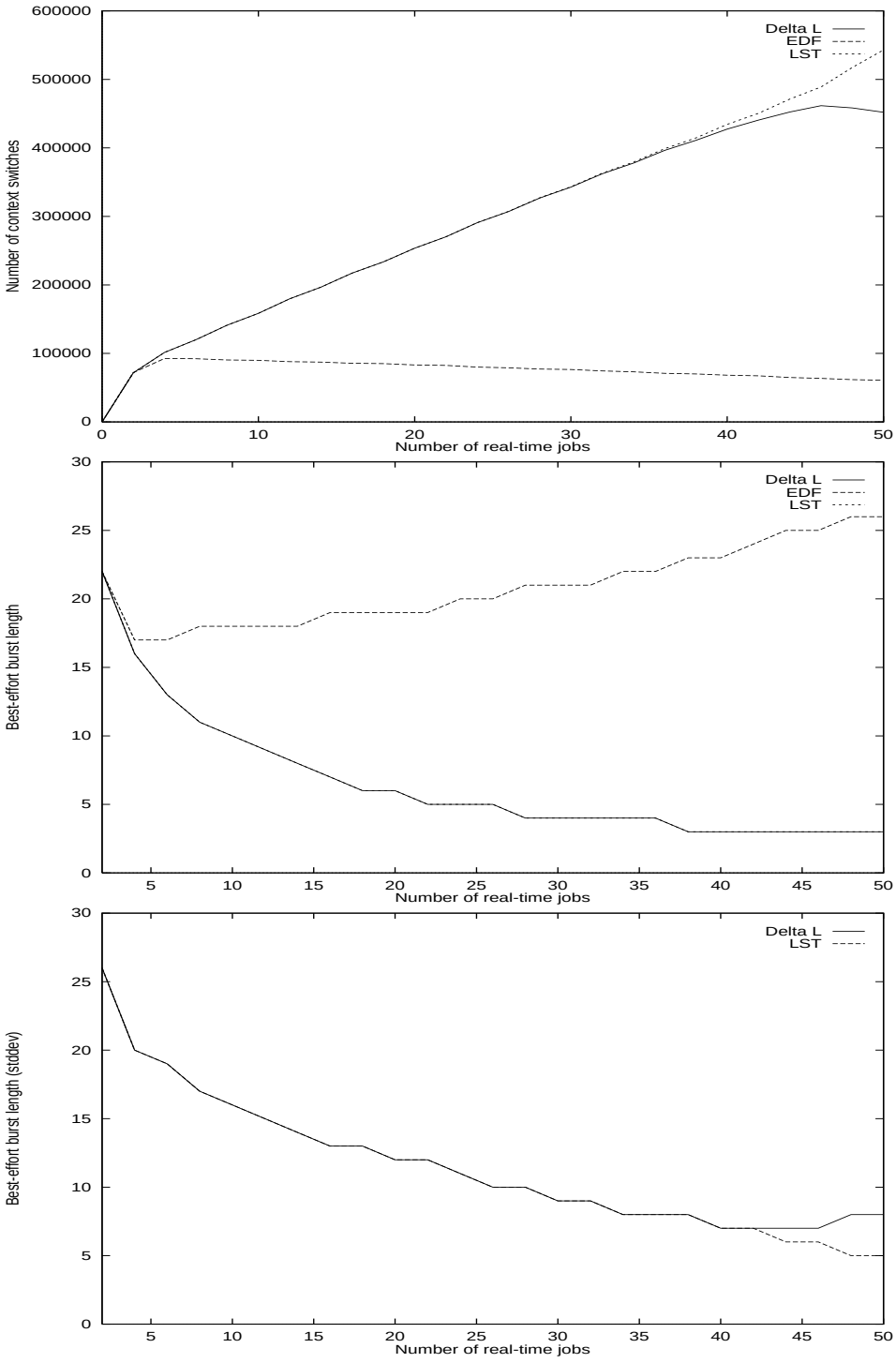

Figure 6.6: Number of context switches between real-time and best-effort dynamic partitions (top), mean best-effort burst lengths (middle) and standard deviations to the mean for the ΔL and LST scheduler (bottom). Note that both the ΔL and LST scheduler have similar mean best-effort burst lengths.

all real-time requests when ΔL_r is exhausted.

Also, since the LST scheduler cannot guarantee that it meets all of the deadlines *and* the best-effort latency increases quickly when peak best-effort load is higher than the available slack time, the LST scheduler is inadequate for real-time disk scheduling.

6.5.2 Random layouts

As is shown in the previous section, seek times play an important role when the amount of slack time is insufficient to execute all of the best-effort requests. In the earlier described simulation, it is assumed that related blocks are allocated close to each other. This means that the dynamic partitions that hold the continuous-media data are sequentially stored on disk. The CELLO disks are stored in dynamic partitions and are packed in a single sequential disk range. The continuous-media dynamic partitions themselves are allocated sequentially on another part of the Quantum Atlas-II disk.

To measure the influence of the seek operations on the overall performance, an experiment is performed where dynamic partition blocks are randomly permuted across the physical disks. Figure 6.7 shows the performance implications of this allocation policy on the performance. First, it shows that when blocks are allocated randomly on a disk, there is a chance that blocks are allocated on the slowest zone on the disk. To admit streams, Clockwise uses the worst-case request timings, so all streams are admitted at the request speeds of the slowest zone. Consequently, fewer streams can be admitted.

All scheduled disks show that for a medium real-time load, LST scheduling performs better than ΔL scheduling, just like in the earlier experiments. However, during high real-time loads, the performance of the LST scheduler is roughly the same as, or better than, that of the ΔL scheduling. The reason for this is that it does not matter anymore to execute all of the real-time requests in a single burst because the disk needs to perform seek operations anyway.

In any case, the best-effort performance of a system where blocks are allocated randomly is substantially less than a system where blocks are allocated close to each other. Figure 6.8 shows the latency differences for CELLO disk 0 between randomly assigned dynamic partition blocks and sequentially assigned blocks when the requests are scheduled by a ΔL scheduler. Clearly, it is important to allocate blocks close to each other.

As described earlier, when blocks are allocated randomly across the disk, real-time streams are admitted at worst-case (*i.e.*, inner zone) service times. In reality this means that for many disk requests the service-time prediction is an over-prediction. Another experiment measures the influence of executing best-effort requests in the over-predicted time. When the scheduler detects that a block is requested on the outer zone (*i.e.*, the disk's fast part) of a disk, it schedules best-effort requests before the real-time request if that best-effort request 'fits' in the over-predicted time.

Figure 6.9 presents the performance differences in best-effort latencies for CELLO disk 0 for all schedulers. The top figure shows that there is no difference in performance for the EDF scheduler. Since best-effort requests are executed only when all real-time requests are finished, only the actual service times of real-time requests are important and the optimization does not help. For both other schedulers, there is a small difference for medium to high loads. The reason

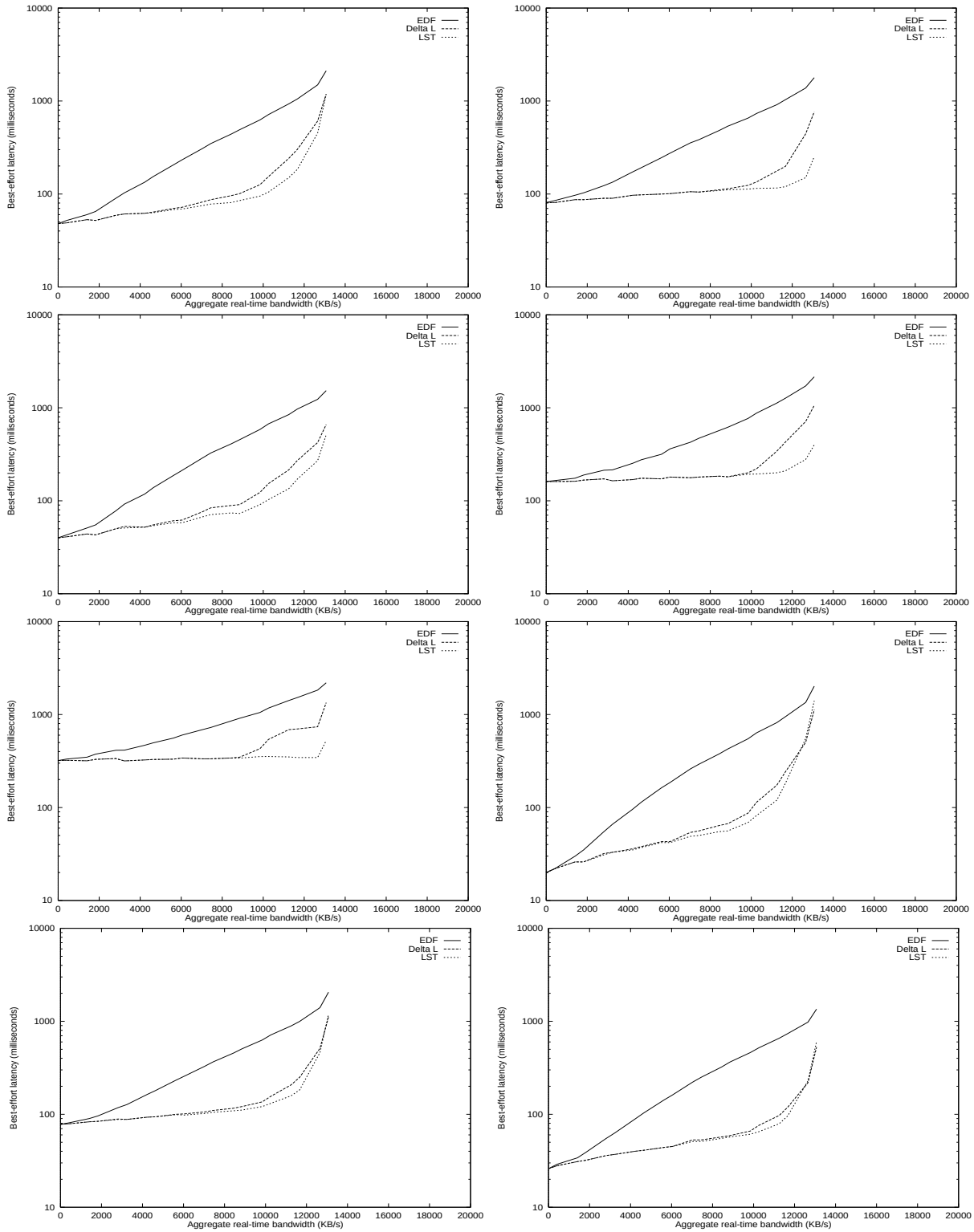

Figure 6.7: Best-effort latencies per HP traced disk when blocks are allocated randomly on disk. Upper left is CELLO disk 0, bottom right is CELLO disk 7.

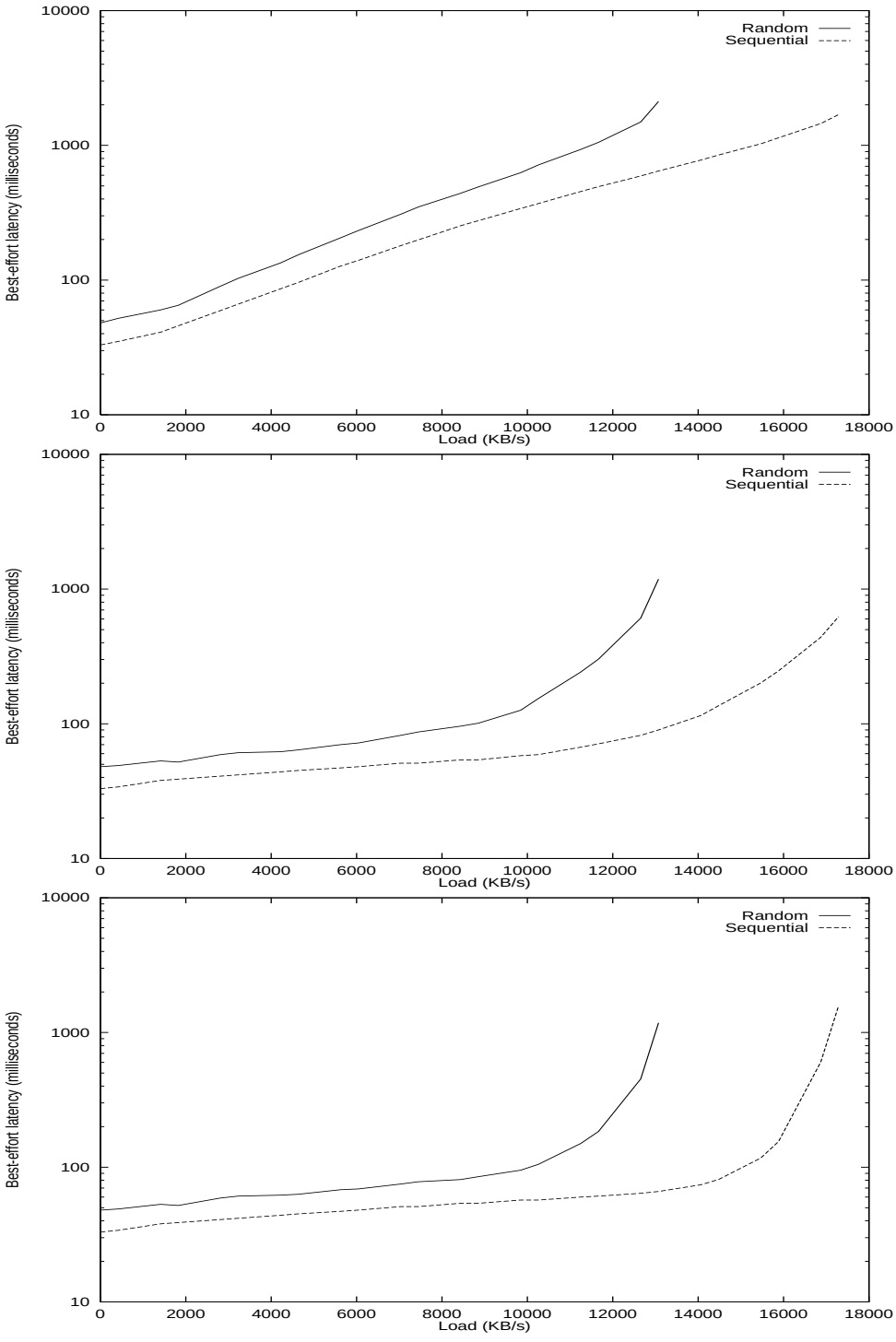

Figure 6.8: Best-effort latencies for randomly- and sequentially-allocated dynamic partition blocks for CELLO disk 0. From top to bottom: EDF scheduling, ΔL scheduling and LST scheduling.

for this is that when there is a high best-effort load, and there is too little time to execute the best-effort requests, finding some slack time does not really reduce the best-effort latency.

6.5.3 ΔL scheduling and memory usage

The ΔL scheduler only guarantees that it meets a deadline when a request is released a full period before its deadline. Given that Clockwise needs in the worst case two transfer buffers for each stream – one that is kept busy by, for example, a network and one that is kept busy by Clockwise – the total memory requirement of all streams is given by:

$$M = \sum_{j=1}^{n} 2D_j b_j$$

where M is the total memory requirement, n represents the number of clients, D_j represents the number of disks on which a dynamic partition is stored that is accessed by client j and b_j represents the block size that is used by client j. Only when disk and network are perfectly synchronized by an application, memory usage M_j for application j is $M_j = (D_j + 1)b_j$.

In the earlier simulation each dynamic partition is stored in a parallel fashion on all disks. This means that in order to send only a single 'Hunt for Red October' audio and video track, a total of 7.5 MB of memory is used. The transmission of 12 'Hunt for Red October' movies (audio and video streams) and 13 'Wallace and Grommit: A Close Shave' audio and video streams consume 129.0 MB of memory space. This real-time load represents approximately 17 MB/s, the point at which best-effort traffic is severely delayed.

An alternative way to deliver 12 'Hunt for Red October' movies and 13 'Wallace and Grommit: A Close Shave' movies is by carefully assigning the continuous-media dynamic partitions to the disks. If, for example, Quantum Atlas-II disk 0 and 1 are used to each deliver 6 'Hunt for Red October' movies and the third disk is used to deliver 13 'Wallace and Grommit: A Close Shave' movies, each disk has roughly the same load as in the 'rotation' experiment. However, the memory usage is reduced by two-thirds because all audio and video files are now served from single disks. To obtain roughly the same aggregate real-time bandwidth, the 'Hunt for Red October' video file is duplicated to two disks.

To measure the effects of the new data assignment, a simulation is performed and the best-effort latencies are measured again against the aggregate real-time load. Figure 6.10 shows the measured latencies of all CELLO disks when they are played concurrently with a real-time load.

The new data assigned shows two results. First, the maximum real-time load is less than is the case for the 'rotating' assignment because simulated disk 0 and disk 1 are assigned approximately 7 % more load than in the 'rotating' assignment. This phenomenon is also shown in Figure 6.4: beyond an aggregate load of 15.5 MB/s, the task set becomes unschedulable.

The second difference between the 'memory-optimized' and 'rotating' assignment is that the performance differences between the LST scheduler and ΔL scheduler are less pronounced. This is because there is ample slack time (simply because there is less real-time load) on the disks to schedule the best-effort requests. Only the best-effort requests that are served by disk 0 and 1 suffered because of the higher load of these disks: their request latencies deteriorate.

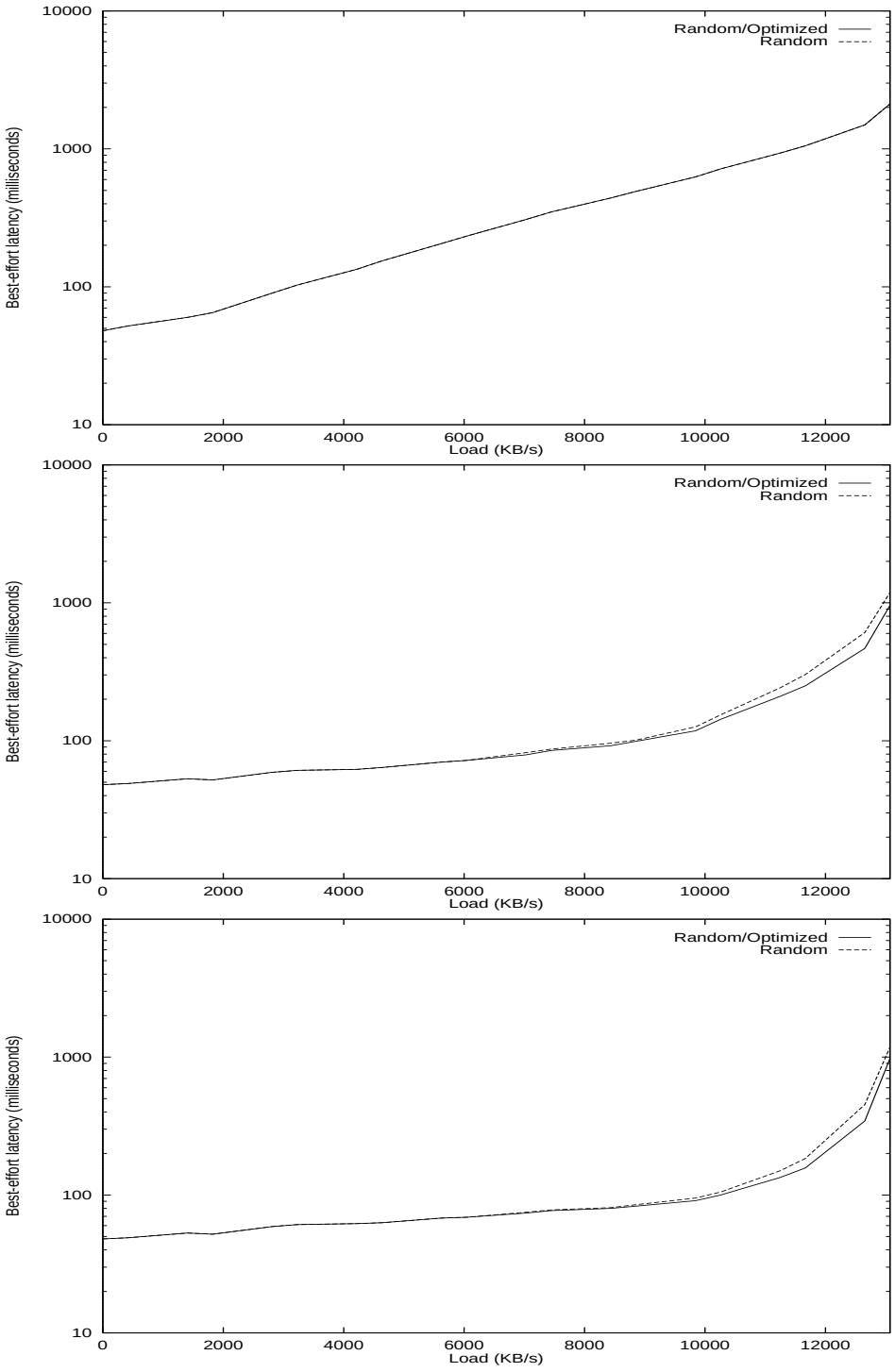

Figure 6.9: Best-effort latencies for randomly allocated dynamic partition blocks for CELLO disk 0 with and without best-effort request optimization. From top to bottom: EDF scheduling, ΔL scheduling and LST scheduling.

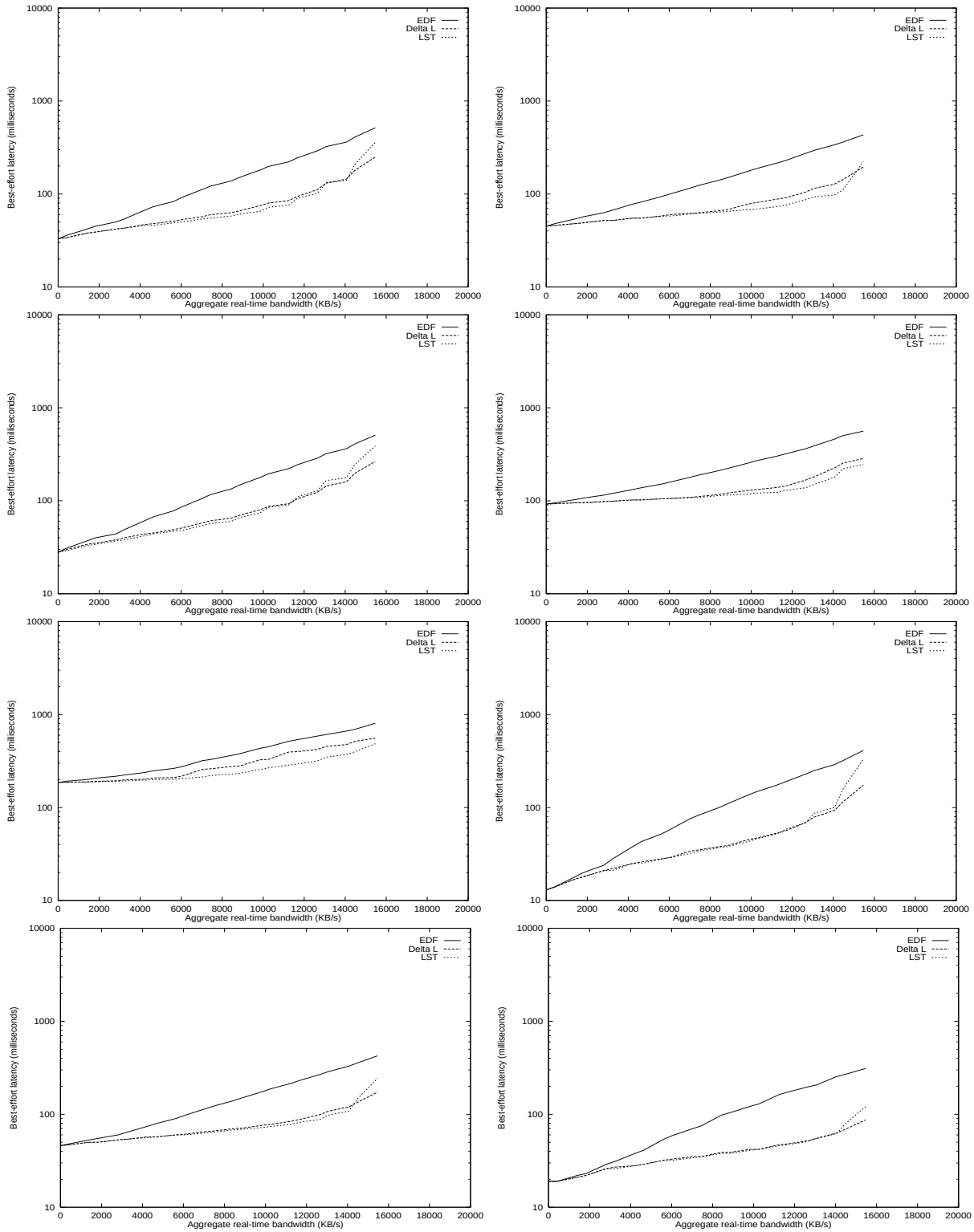

Figure 6.10: Best-effort latencies per HP traced disk (memory optimized). Upper left is CELLO disk 0, bottom right is CELLO disk 7.

The ‘memory-optimized’ experiment is re-executed, but with twice as large memory buffers to measure the effects of larger values for ΔL. Instead of using a 1 MB block size for the movie ‘Hunt for Red October’, this movie is played with 2 MB blocks and all of the audio streams are assigned 512 KB blocks instead of 256 KB blocks. The effect of the larger buffers on ΔL is shown by the curve labelled ‘Memory optimized (large buffers)’ in Figure 6.4. The precomputed slack time nearly doubles in most cases, and the amount of memory that is in use for 12 ‘Hunt for Red October’s and 13 ‘Wallace and Grommit: A Close Shave’s increases to about 80 MB of memory space. Also, the schedulability of the disks is improved, which means that the disks can sustain a higher real-time load.

Figure 6.11 shows the best-effort latencies versus the aggregate real-time bandwidth in the ‘memory-optimized’ block assignment with large blocks. It shows that all scheduling effects are similar to what happens in the earlier described ‘rotating’ block assignment. When there is a high real-time load, there is little slack time available and when requests are scheduled by a LST scheduler, lots of precious time is wasted on disk seek operations. In all cases the measured best-effort latencies for the LST scheduler are almost equal to or worse than the measured best-effort latencies for extreme loads when requests are scheduled by an ΔL scheduler. Also, the measured best-effort latencies in the ‘memory-optimized’ assignment with large transfer buffers are lower than the measured best-effort latencies when small transfer buffers are used.

Memory bandwidth can be traded against disk schedulability. When larger buffers are used, more slack time is available to admit new jobs and to schedule best-effort requests.

6.5.4 Best-effort QoS

It is not strictly necessary to execute best-effort requests only in slack time. Especially when the aggregate real-time load increases, it can be important to associate QoS parameters to best-effort requests. If, for example, a best-effort application requires operations to complete in a certain pace, assigning a QoS parameter in the form of a period and a service time, similar to USD’s approach, instructs the disk scheduler to schedule the best-effort application like a real-time application.

One approach to schedule best-effort tasks uses a simple periodic real-time server and the ΔL scheduling algorithm. Every scheduled best-effort task is assigned a periodic server that acts as a container for the best-effort requests from that task. The periodic server is used to assign QoS scheduling parameters in the form of release times and deadlines to the queued (and scheduled) best-effort requests. Also, even though the best-effort requests are assigned release times and deadlines through the periodic server, it is still advantageous to prioritize best-effort requests. Hence, best-effort requests can still be executed by the slack time stealing algorithm in ΔL time.

If too short periods are used by the periodic server, ΔL may become too small to schedule other real-time tasks. From a disk performance perspective, it is advantageous to use large buffer transfers for bulky continuous-media data transfers. When such bulky requests are scheduled concurrently with short period periodic servers, the schedulability of the disks quickly decreases. Therefore, a best-effort application can only schedule n requests per period with a relatively long period. Since it is difficult for the user to determine the worst-case service times for n requests precisely, a user simply presents a period and a slice that it expects to use for n requests.

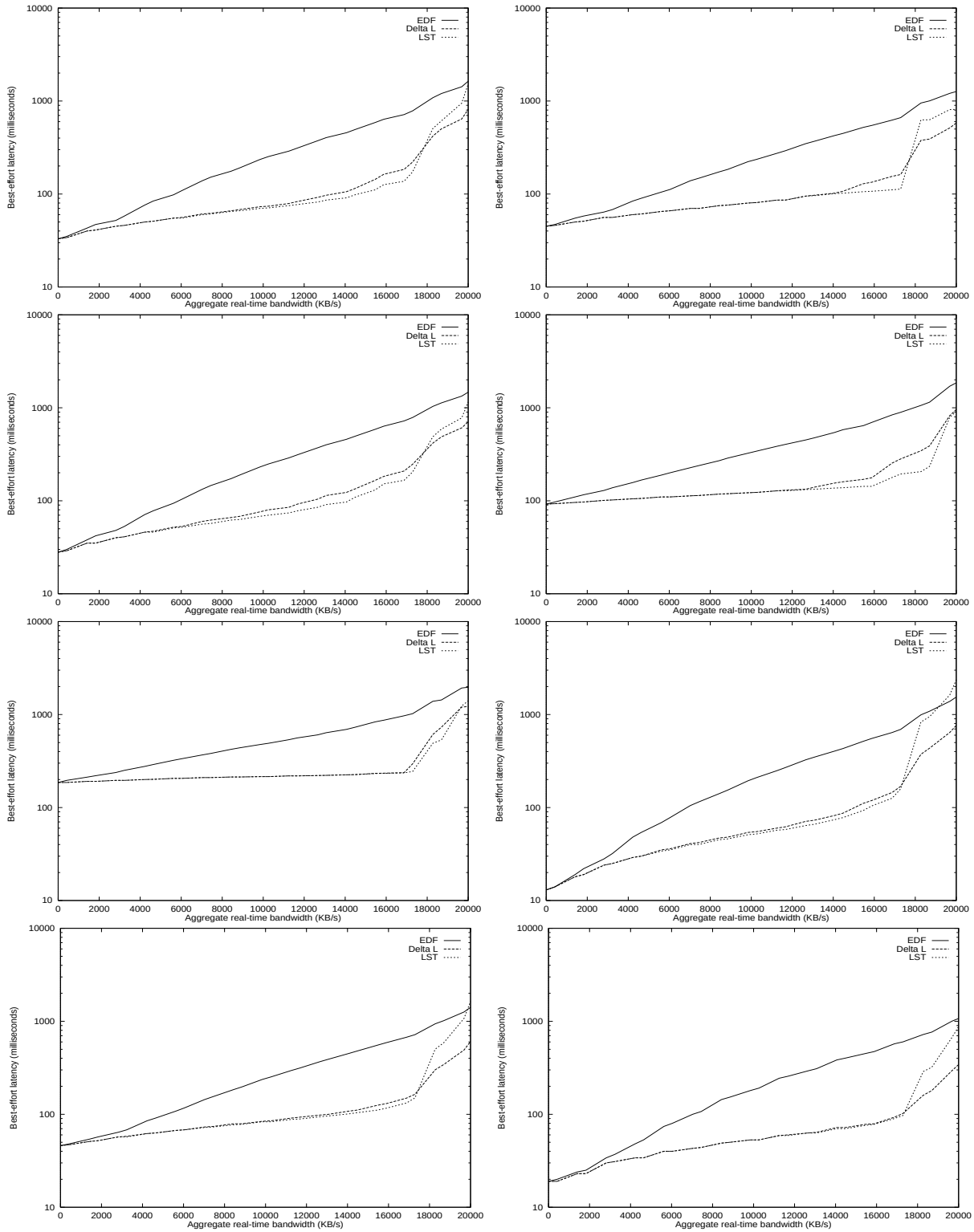

Figure 6.11: Best-effort latencies per HP traced disk (memory optimized – large buffers). Upper left is CELLO disk 0, bottom right is CELLO disk 7.

Clockwise deals with best-effort QoS parameters in a way similar to other real-time QoS requests. The requested service time and period together with the QoS parameters of all other real-time tasks are analyzed by Clockwise's schedulability test. When the task set including the best-effort periodic servers is schedulable, the best-effort application's periodic server is admitted as a new real-time task. Whenever a request arrives from a scheduled best-effort task, Clockwise calculates a release time and a deadline based on the QoS parameters of its periodic server. When there are more best-effort requests available than can be coped with by the periodic server in a single invocation, release times and deadlines of next containers are used. Such requests are then scheduled to be executed in the future.

When the ΔL scheduler uses precomputed slack time to give precedence to scheduled best-effort requests the latencies of such scheduled best-effort requests decreases. When a scheduled best-effort request is executed in ΔL slack time, all release times and deadlines of other queued scheduled best-effort requests from the same task need to be recalculated: their release times and deadlines may have shifted to earlier times.

To analyze the effects of assigning a QoS contract to a best-effort domain, a single disk from the HP traces (CELLO disk 5) is assigned a period and a service time. The latencies are measured of all disks including the scheduled disk 5 in parallel with 24 combined audio and video streams with an aggregate real-time load of 16.5 MB/s. The real-time data is assigned to the disks in a 'rotating' manner.

Figure 6.12 presents the number of I/Os that took place on disk 5 of the CELLO trace sometime during the day of June 1st, 1992. This day and period are chosen because during this period, disk 5 endured a large number of I/Os. The horizontal axis shows the time of day measured in seconds, the vertical axis shows the number of I/Os per second. Figure 6.12 shows that a period and service time setting that allows between 20 and 40 I/Os per second optimizes for this situation.

Figure 6.13 presents the mean and maximum I/O latencies of all requests throughout the selected period[11] when disk 5 is given three different QoS settings: 0 %, 10 % of the available disk time (*i.e.*, 100 ms per second) and 20 % (*i.e.*, 200 ms per second).

Clearly, not negotiating a QoS contract for the best-effort traffic leads to high I/O latencies. Consider, for example, $t = 965$: at this particular time there is not enough slack time available to schedule the unscheduled best-effort traffic with the real-time requests and best-effort latencies are high.

When a QoS setting is assigned to CELLO disk 5, the performance of the scheduled best-effort domain quickly improves. The mean latencies decrease to approximately 1 second, the period of the QoS setting. Only in those cases that more requests are queued than can be serviced in the allocated service time, queues build up again and requests need to wait for next container periods. The figures also show that when there is a substantial best-effort load for CELLO disk 5, the QoS setting of 20 % performed (slightly) better than an allocation of 10 %. During these periods more scheduled best-effort requests are serviced in their slice.

An interesting peak is shown $t = 940$ in Figure 6.13. At this point the latency of unscheduled best-effort requests is better than for both scheduled cases; in fact, the 20 % assignment leads

[11] Note the Figure 6.12 presents seconds, while the Figure 6.13 presents the time in minutes.

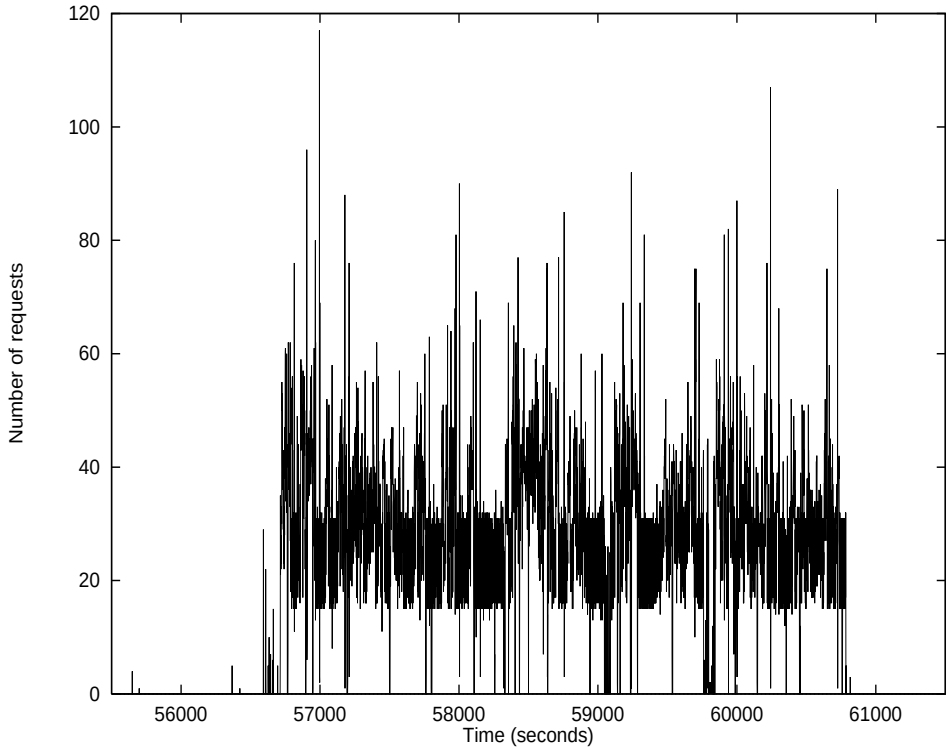

Figure 6.12: Number of I/Os per second for a period on June 1st, 1992 on CELLO trace disk 5.

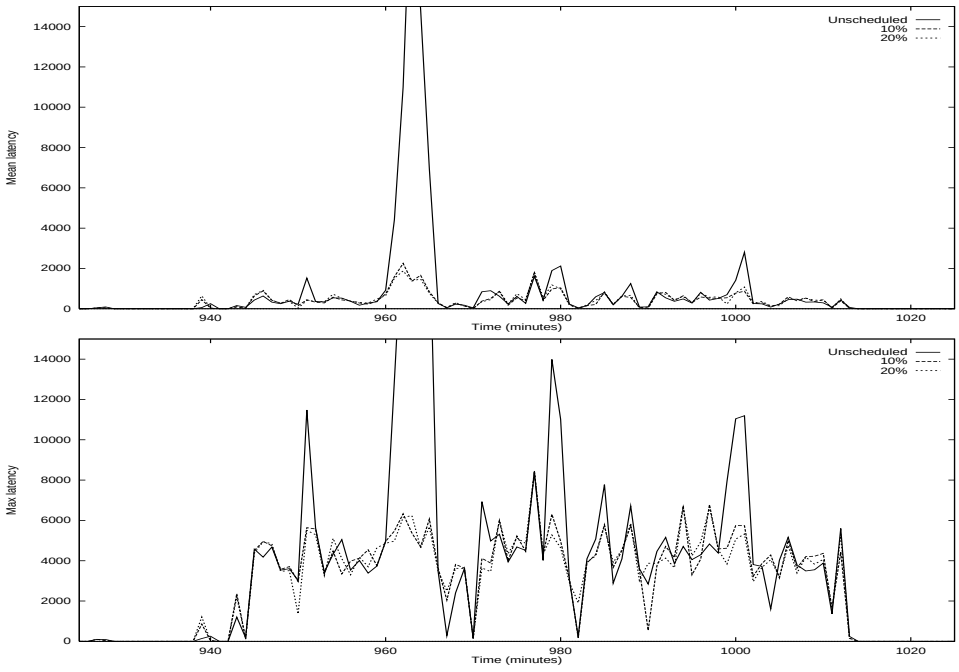

Figure 6.13: Mean (top) and maximum (bottom) measured latencies (in milliseconds) for a scheduled and unscheduled CELLO disk 5.

to worse best-effort request latencies when compared to a QoS setting of 10 %. The reason for this is that the available slack time ΔL decreases when there is more scheduled I/O load: in the unscheduled run ΔL is 925 ms, in the 10 % assignment ΔL is 825 ms and for the 20 % assignment ΔL is 725 ms. Smaller values for ΔL implies that the slack time is exhausted sooner and more best-effort requests need to wait longer until ΔL_r is replenished again. Since the ΔL scheduler also considers scheduled best-effort requests (*i.e.*, best-effort requests with a release time and deadline) that are scheduled to be executed in the future, a smaller ΔL means that fewer scheduled best-effort requests are executed with priority. Hence, such requests have to wait either for their original execution time as is dictated by EDF order, or for ΔL_r to be replenished.

6.5.5 Quality of Service crosstalk prevention

The prevention of QoS crosstalk is important to guarantee timely delivery of continuous-media data to 'behaving' applications. QoS crosstalk is the phenomenon that non-behaving applications can have on applications that behave according to their QoS contract.

EDF schedulers have an effective way of preventing QoS crosstalk. Each application informs the scheduler on beforehand of the service time and the period with which it requests service. These values are used by the scheduler to associate a release time and a deadline to requests as is described earlier. A non-behaving application (*i.e.*, a task that issues requests too quickly compared to its QoS contract), is punished by being scheduled to execute in the future (*i.e.*, its release times and deadlines are set to the future): as long as the requests are not due and there is no slack time in the schedule, the non-behaving applications are simply not serviced.

'Behaving' real-time applications are not hindered by the 'non-behavior' of other real-time applications. The only applications that are hindered by non-behaving real-time applications are unscheduled applications such as best-effort applications. Since non-behaving applications can effectively allocate all of the remaining slack time in a schedule, best-effort applications can be denied any service by Clockwise.

To measure the effects of QoS crosstalk prevention, an experiment is performed where all best-effort domains are assigned a small fraction of the available resources and are run simultaneously with a non-behaving continuous-media task. To limit the amount of simulations, only a single day of workload is simulated (June 1st, 1992). Each CELLO disk allocates a service period of 1 second for every period of 30 seconds per disk.

The continuous-media task allocates 1 MB/s of service and it sends disk requests of megabyte blocks each to the simulated disks at a fixed rate of initially 1 MB/s. For each simulation run, the *inter-block* period, the time between two successive real-time requests, is slightly decreased to the point that the continuous-media task sends disk requests back-to-back to the simulated disks; the latter happens when there is an *over-allocation* of a factor 25; the 'misbehaving' continuous-media application is requesting 25 MB/s. Initially, the inter-block period is precisely 1 second.

Figure 6.14 shows the results of two simulations. First, the figure curves that are labeled 'RT' are the measured mean latencies for the scheduled best-effort domains in relation to the over-allocation of the non-behaving task for all 8 CELLO disks. The figure clearly shows that best-effort latencies increase when the non-behaving task consume more resources, but at some point the over-allocation of the misbehaving task does not lead to extra delays in the scheduled

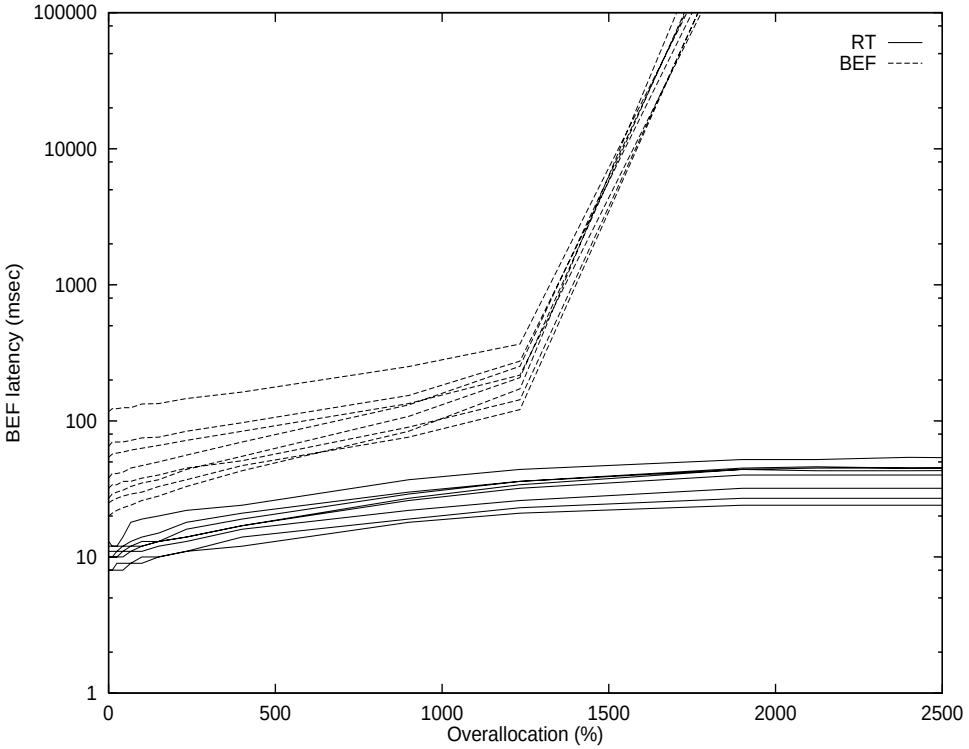

Figure 6.14: Mean scheduled and unscheduled best-effort latencies versus the over-allocation of a non-behaving tasks. RT stands for real-time, BEF stands for best-effort.

best-effort domains: the best-effort latency stabilize at approximately 25–51 ms.

The figure lines that are labelled 'BEF' are the measured latencies for unscheduled disks in comparison with a single non-behaving task. The figures clearly show that best-effort domains can better be scheduled with a QoS contract: measured latencies quickly increase once these requests really have to compete with the non-behaving task.

Figure 6.14 also shows a large difference in latency between scheduled and unscheduled best-effort traffic when there is little real-time traffic. The difference is caused by busy best-effort periods with little ΔL_r slack time. When best-effort traffic is scheduled, requests do not have to wait for ΔL_r to be replenished. Instead, requests are executed by means of their release time and deadline.

6.6 Summary

This chapter presents a novel way to schedule best-effort requests and real-time requests on a disk in such a way that the latencies of the unscheduled best-effort requests are minimized, while not missing any of the deadlines of the real-time requests. The reason this is important is because it is quite likely that best-effort traffic contains synchronous requests on which users are waiting for their completion. As long as deadlines are not missed, it does not matter when the real-time requests are executed.

The scheduling techniques that are presented in this chapter are based on earlier work by Jeffay *et al.* who have introduced a schedulability analysis for a nonpreemptive *Earliest Deadline First* (EDF) scheduler. When a task set passes Jeffay *et al.*'s schedulability analysis, a nonpreemptive EDF scheduler can schedule requests from the task set without missing deadlines. Such a scheduler is in particular important to schedule disks simply because once a disk request has started, it is impossible to preempt it.

The schedulability analysis of Jeffay *et al.*, is extended to pre-calculate schedule slack time. It is proven that this slack time is available between the end of *any* invocation of a task and its deadline. Given that all real-time requests finish at least this slack time (ΔL) before their deadline, all real-time requests can also start at most ΔL later than is originally planned. This slack time is used to schedule best-effort requests to minimize their latencies.

The ΔL scheduler is compared with a standard EDF and a *Latest Start Time* (LST) scheduler. The former scheduler schedules best-effort traffic *after* the real time traffic, the latter schedules the best-effort traffic before the real-time traffic. It is shown that the standard EDF scheduler behaves poorly: best-effort latencies quickly increase to unacceptable latencies once the aggregate real-time load increases.

When best-effort requests are scheduled by a LST scheduler, the measured best-effort latencies are slightly better when compared to the latencies when requests are scheduled by a ΔL scheduler for light to moderate aggregate real-time loads. The reason for this is that the LST scheduler decides at a later instance to schedule real-time load, so that more best-effort requests are executed prior to real-time requests. For high real-time loads, the LST scheduler schedules best-effort traffic worse than the EDF and ΔL scheduler. Although the LST scheduler schedules the real-time traffic at the latest possible instant if there is unscheduled best-effort traffic avail-

able, the cost of performing disk context switches between the best-effort and real-time dynamic partitions becomes prohibitively high: the scarce slack time is used inefficiently by long seek operations to perform the disk context switches.

It is also shown that existing deadline-dynamic disk-scheduling techniques such as USD's EDF scheduler and Symphony's LST scheduler cannot guarantee real-time deadlines. The reason for this is that these schedulers do not account for the nonpreemptiveness of a disk. Clockwise's disk schedulability analysis takes this into account and guarantees deadlines: it schedules the disks through a real-time disk scheduler.

Lastly, it is shown that an EDF scheduler has a simple and effective way to prevent QoS crosstalk. By assigning release times and deadlines according to the QoS contract instead of the applications actual behavior, 'misbehaving' applications are scheduled to be executed in the future; they are assigned future release times. Clockwise's ΔL scheduler only schedules requests with future release times through a best-effort class of service.

Chapter 7

Performance evaluation

The performance of the storage applications that run on a Clockwise server is dominated by the way the disk scheduler schedules the disk requests. In Chapter 6 a number of performance experiments are described that give insight in the capabilities of a mixed-media storage system and in particular the influence of the disk scheduler on the overall performance. By showing a similarity between the simulated performance numbers and actual performance numbers, conclusion that are drawn from the simulations are validated for a real system. This chapter presents such measurement comparisons.

When the simulated performance numbers are validated, there is no need for extensive measurements in a real Clockwise. Instead, only a few detailed actual measurements are required to measure those performance issues that have not or can not be measured by a simulator. By combining the detailed real measurements with the (validated) simulation results a detailed insight is given in the actual behavior of a Clockwise system.

There are several detailed measurements and experiments that are executed on a real Clockwise. Of these detailed experiments, an experiment called ‘service-time prediction’ is the most important one. Clockwise uses the raw performance numbers that are described in Chapter 4 as a basis for request service-time prediction. However, these measurements only measured a limited set of block sizes and disk addresses to learn of the performance of a disk. Clockwise, on the other hand, allows the reading and writing of data blocks of any size and at any location on disk as long as the data is read from or written to a dynamic partition. Measurements are required to learn how the raw performance numbers need to be applied to predict request service times accurately.

One of the design goals for Clockwise is QoS crosstalk prevention. QoS crosstalk prevention is the technique to prevent misbehaving storage applications from using the resources that are meant for behaving applications. To learn how well Clockwise can withstand misbehaving applications, an experiment is performed where an application is instructed to over-use disk resources. The effects on another (behaving) application are measured.

Most performance experiments that are described so far involve the reading and writing of data on Quantum Atlas-II disks. These disks are not really state-of-the-art disks and the new Seagate Cheetahs, for example, perform at a much higher data rate. To learn of the maximum I/O rates that can be delivered by a Clockwise server, an experiment is performed that measures how

well Clockwise deals with a large number of parallel Seagate Cheetah disks. The performance experiments that are performed with the Seagate Cheetah disk are limited and only give insight in the scalability of Clockwise.

So far, only disk performance is used to characterize the capabilities of a Clockwise machine. Another important resource, the CPU, is required to drive the SCSI device drivers and to execute Clockwise itself. If executing Clockwise at full speed implies high CPU loads, the applicability of Clockwise is limited. To learn of the CPU usage, the multi-disk Seagate Cheetah performance experiment is executed such that also insight is given in the CPU usage of Clockwise. For this, the underlying operating system, Nemesis, is altered to collect CPU scheduling statistics during the measurement.

Lastly, Clockwise is a mixed-media storage system, which implies that data that is recorded or played-back from a Clockwise server is transferred across a network. Given that the maximum aggregate performance of a Clockwise machine is (much) more than what can be coped with by the University's network (an OC-3 ATM network), a throughput projection is made when Clockwise is used for recording and playbacks of AVA continuous-media data with faster networks.

7.1 Validating simulated performance results

In Section 6.5 a large number of performance experiments are described with a discrete-event simulator that behaves approximately like a set of parallel Quantum Atlas-II disks. Based on the simulation results general conclusions are drawn for the behavior of the various schedulers in a real system. The reason for performing the scheduling experiments in a simulator is because it is usually much simpler and quicker to execute the performance experiments in an off-line simulator. If performance measurements are performed online, *i.e.*, for real, the measurements also complete in real time. Especially when loads characteristics are not easily defined, such as is the case for UNIX file I/O, long runs are required to learn of the behavior of the system. Retrieving reliable measurement results in a real system, is, in that case, a task of great patience.

Systems are often evaluated in off-line event-driven simulators that run a predefined set of operations and behavioral conclusions are drawn from what is measured in the simulator. A number of other systems have also been analyzed by means of off-line event-driven simulators, such as HP's AutoRAID system [104]. The events that are used for the simulations consist of a log (or trace) of actual recorded disk operations. The traces are replayed on the simulator that behaves much like its real counterpart. However, as Chris Ruemmler already pointed out, it is usually a tedious job to re-create a real system in a simulator [86].

To make sure that a simulator behaves like its real counterpart, a simulator can be integrated in a real system [15]. The problem with this approach, however, is that at some point it is hard to grasp what parts of the system need simulation and which do not – the problem of constructing these systems becomes more of an issue than the actual goal, namely to analyze algorithms.

The most efficient way to learn of the performance of a system is by executing a number of pre-recorded (disk) traces on a system simulator and later to validate the simulated performance numbers through (a limited) set of measurements in a real system. When the performance numbers in the real system correspond to the simulated results, there is a good indication that

Disk	Requests	Average load/s	Peak load/s
0	31066	23.9	130
1	0	0	0
2	1015	0.8	42
3	6	0.0	6
4	6	0.0	6
5	8401	6.5	88
6	12427	9.6	207
7	831	0.6	31

Table 7.1: Load on June 1st, 1992, 17:48:00 to 17:09:40 per disk.

conclusions that are drawn from the simulated results are valid for the real system. The validation process itself is actually quite simple. A part of the events from the earlier recorded traces are replayed on a real system and the timing values are compared to the values generated by the simulator. When the simulator is 'good enough', the values are approximately the same.

The approach to validate the simulated Clockwise performance results from Section 6.5 is by executing a short part from the HP disk traces on a real Clockwise and on the Clockwise simulator together with a real-time load. The reason for using only a small part of the traces is to limit experimentation time and to allow the measurement of all schedulers and load characteristics.

The simulator is organized in a manner that the layout of the dynamic partitions is identical to the organization of data on a real Clockwise machine. The simulator simulates three parallel Quantum Atlas-II disks, and the real Clockwise machine is also equipped with three Quantum Atlas-II disks. The Quantum Atlas-II disks are connected by Fast SCSI-2 buses in both the simulator and the real system.

The selected trace file is a short part from the CELLO disk trace on June 1st, 1992 (16:48:00 to 17:09:40). This day and time are chosen because it is the day and time with the highest load. Throughout the 1,300 second period a total 53,752 requests are executed on the eight CELLO disks. Table 7.1 presents the number of requests per disk, the average load per disk per second and the peak load per disk per second. Since disk 1, 3 and 4 do not contain a significant load during the measurement period they are omitted from the traces.

Figures 7.1 and 7.2 presents the measured and simulated performance numbers for disks 0, 2, 5, 6 and 7 for each of the schedulers. The figures show for all three schedulers (EDF, ΔL and LST) the measured and simulated best-effort latencies with a varying real-time disk load up to 17.5 MB/s.

All of the figures show that the performance numbers from the real system differ only slightly from the simulator, although the overall conclusions from Section 6.5 remain valid. When best-effort requests are scheduled by the LST scheduler, and there is a high real-time load, best-effort request latency is high. When requests are scheduled by the ΔL scheduler, their latencies do increase with increasing real-time load, but latencies do not become worse than latencies of requests that are scheduled by the standard EDF scheduler.

There are several reasons for the measured performance differences between the real and simulated system. First and foremost, the simulator's disk write functions are inadequate: they only

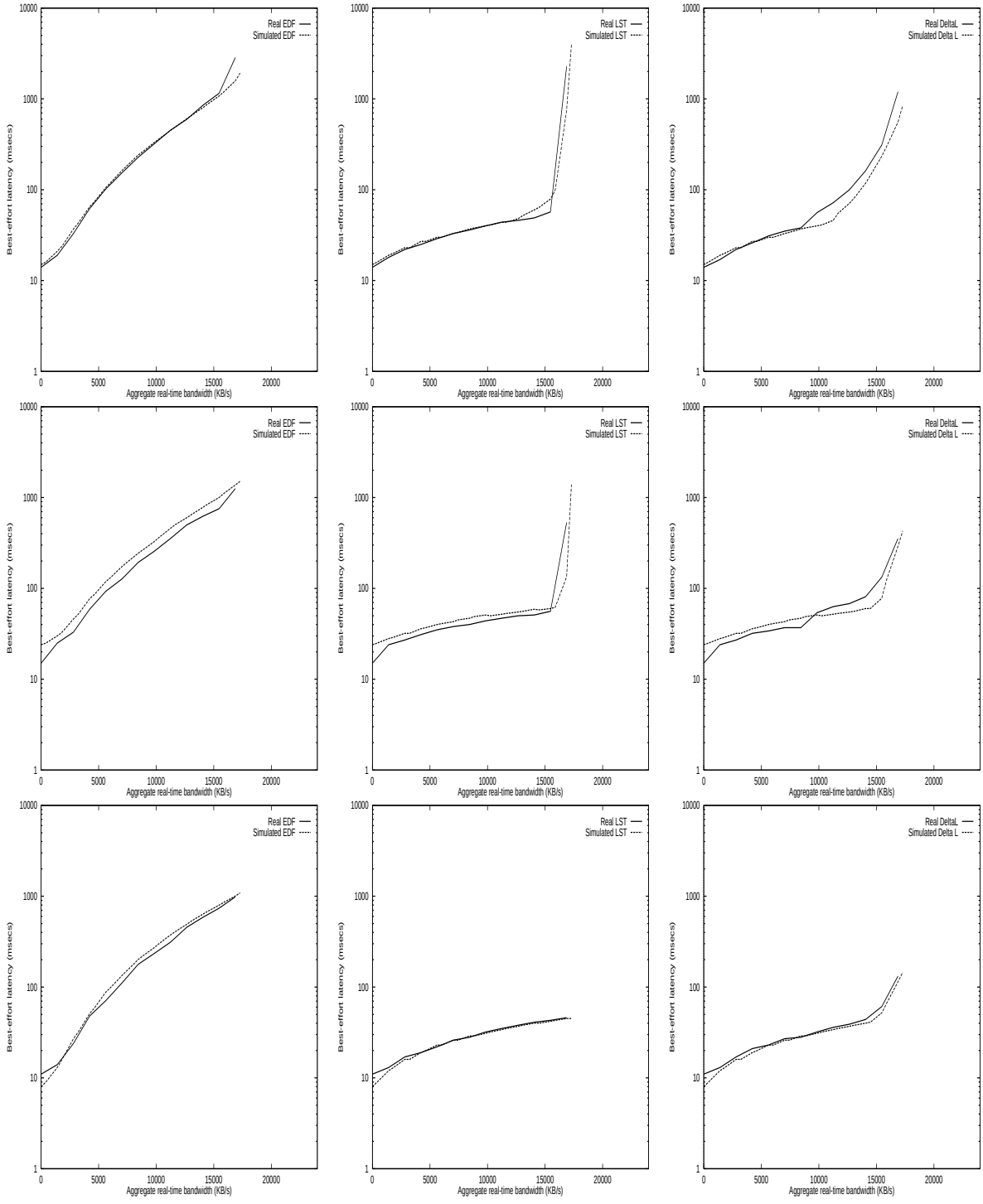

Figure 7.1: Measured and simulated performance for 5 disks. The rows describe the performance of CELLO disk 0, 2, and 5, the columns describe the behavior of the EDF, ΔL and LST scheduler.

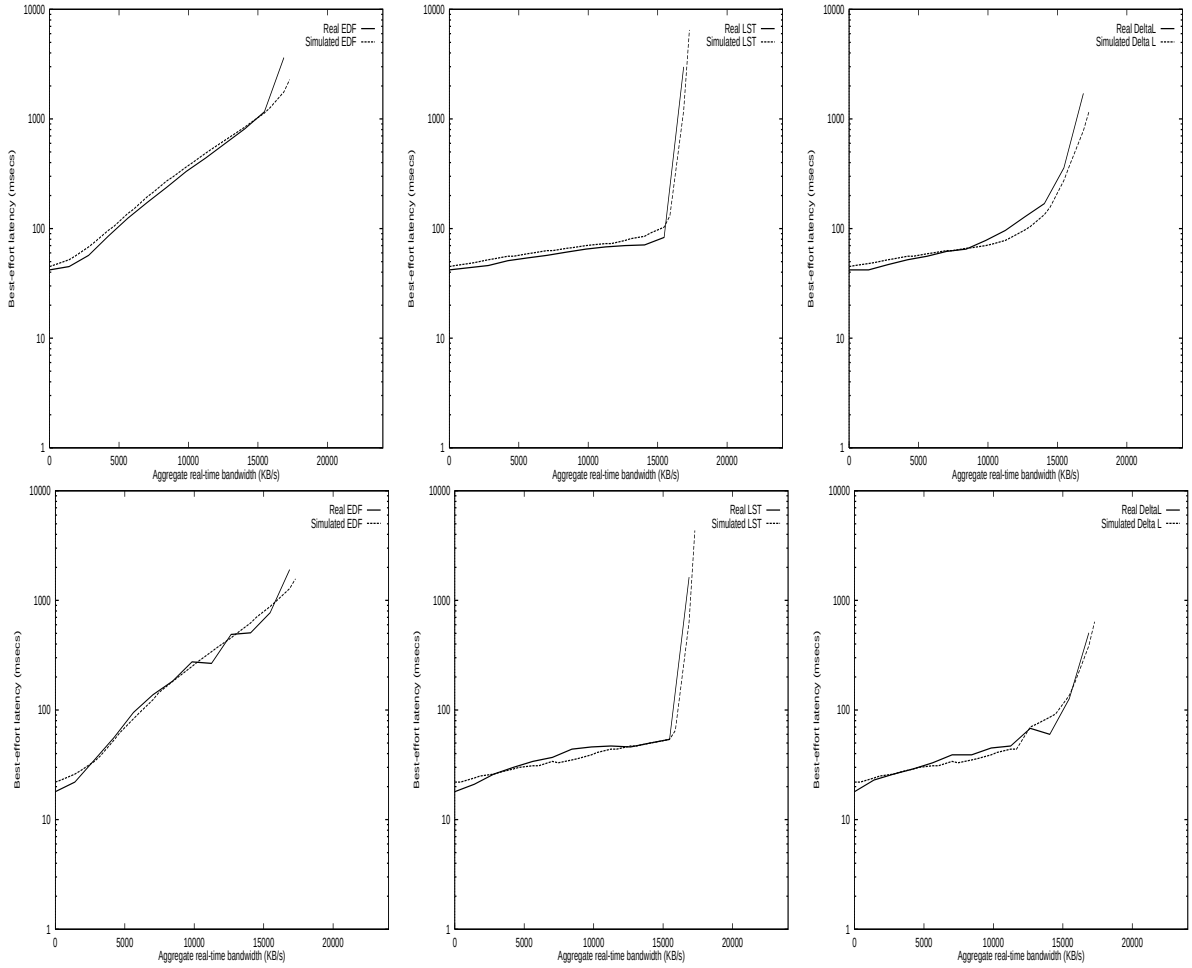

Figure 7.2: Measured and simulated performance for 5 disks. The rows describe the performance of CELLO disk 6 and 7, the columns describe the behavior of the EDF, ΔL and LST scheduler.

use the timings from Chapter 4 instead of actually simulating all of the mechanical and electrical steps that are performed inside a disk. This means that write operations can have different service times than what is measured on a real disk. The read functions, on the other hand, do model all of the disk steps. Also, the simulator does not model operating-system overhead. Instead, the simulator adds a fixed scheduling overhead to all of the operations so that the simulated read performance corresponds to the actually measured read operations.

Given time, the simulator can be made such that it models a Quantum Atlas-II disk perfectly. One can wonder if this is a sensible approach. The performance measurements on a real system show that the conclusions from the simulations can also be applied on a real system. By showing that the real system behaves in a similar manner as the simulator, the simulated results are validated. A perfect disk simulator will not change the conclusions, and can, therefore, be considered a waste of time.

7.2 Service-time predictions

The Clockwise disk scheduler relies on good service-time predictions. If the service time predictions are such that all requests that are executed on Clockwise complete in less time than is predicted, Clockwise guarantees that all deadlines are met. If, however, the actual service time is higher than what is predicted, it is possible that tasks miss their deadlines.

The current implementation of Clockwise uses the measurements from Chapter 4 as a basis for service-time predictions. When an application presents QoS parameters to Clockwise, Clockwise predicts the service time based on service-time tables that are generated by the raw performance measurement application. The problem with this approach is that only when Clockwise runs in precisely the same configuration as the configuration that is used to measure the raw performance, the raw performance numbers can be used directly in Clockwise. If there are hardware differences, or when there are data layout differences, the raw performance numbers need to be adjusted to the new situation.

To make sure that there are no hardware differences between the earlier described measurement machine and the Clockwise server, the same machine is used for executing Clockwise. The earlier raw-performance measurements are performed on a 200 MHz Pentium-Pro based machine that is equipped with a maximum of four Quantum Atlas-II disks[1], which are connected by NCR53c810a and NCR53c875 SCSI/PCI controllers.

7.2.1 Read performance

To validate read service-time predictions, a number of read experiments are performed and the measured service times from Clockwise are compared to the raw performance service times. For this a single dynamic partition is created on each of the disk zones on a Quantum Atlas-II disk, and the service times are measured by reading blocks sequentially from disk. Given that there

[1]The maximum number of Quantum Atlas-II disks that were available for the Clockwise measurements was only 3. Between the date of raw-performance measurements and Clockwise measurements, the 4th disk had been confiscated and is used in a production file server.

is only a single measurement application, only the request for block 0 involves a seek operation. The remainder of the requests are executed sequentially.

Figure 7.3 and Figure 7.4 presents the cumulative distributions of measured service times both in the raw case and in the Clockwise case for various block sizes in zone 0 and zone 11 of the Quantum Atlas-II disk. The following block sizes are measured: 1 KB, 16 KB, 64 KB, 256 KB, 512 KB and 1 MB. The reason why only zone 0 and 11 are presented is for brevity and because they present both ends of the performance spectrum of the Quantum Atlas-II disk: zone 0 is the fastest, zone 11 is the slowest.

In all cases, it shows that the maximum service time in Clockwise is exactly one rotational delay longer than the raw performance measurements. The reason that Clockwise is one rotation slower is because dynamic partition blocks are not aligned to track boundaries. Consider, for example, dynamic partition block assignments for zone 0. There are 179 sectors per track for this zone on a Quantum Atlas-II disk, which means that most sector numbers within a track are used 8 or 9 times as start sector of a megabyte dynamic partition block. This implies that each start track of a dynamic partition block contains, on average, half a track worth of data that belongs to the previous dynamic partition block. Hence, reading block n of a dynamic partition block also involves reading, on average, half of last track of block $n - 1$, *i.e.*, this data is read twice when a dynamic partition is read sequentially.

The reason there is not a large performance difference in short block service times even though dynamic partition blocks are not aligned to track boundaries, is because the disk still waits until all the data is read into the disk cache before it is sent to the host. The only performance difference occurs when a Clockwise short block is located on two heads or two tracks. In this case the time to perform a head or track switch is included in the transfer time. This time is not included in the raw performance measurements.

All figures show a long tail in the distribution, which is caused by bad or errors blocks on the measurement disk. When the disk finds a bad block on disk it re-maps the bad blocks to spare blocks somewhere else on disk. When the disk finds an error, it retries one or more times. In any case, the operation takes longer to complete. The reason this phenomenon does not show up in the raw performance measurements is because the raw performance measurements only uses a single disk area to measure the performance. This raw measurement area does not contain bad or error blocks. How Clockwise deals with bad blocks or disk errors is explained later in this section.

The unalignment of dynamic partition blocks implies that for the service-time prediction, a full-rotational delay and a head and cylinder switch needs to be added to the predicted service times, which reduces the real-time schedulability of a disk. The reason that a full rotation needs to be added to the predicted service time is because the first sector of a dynamic partition block can be located at the last sector of a track.

A possible solution is to align the dynamic partition blocks on track boundaries. This solution is debatable: from a performance perspective it is certainly true that a Clockwise with aligned blocks performs better than a Clockwise without aligned blocks. However, such a solution limits Clockwise's ability to freely move blocks within the array of disks. Also, some Clockwise storage applications know of the actual size of a dynamic partition block and optimize for this. When dynamic partition blocks have a variable size it means that these applications need to be

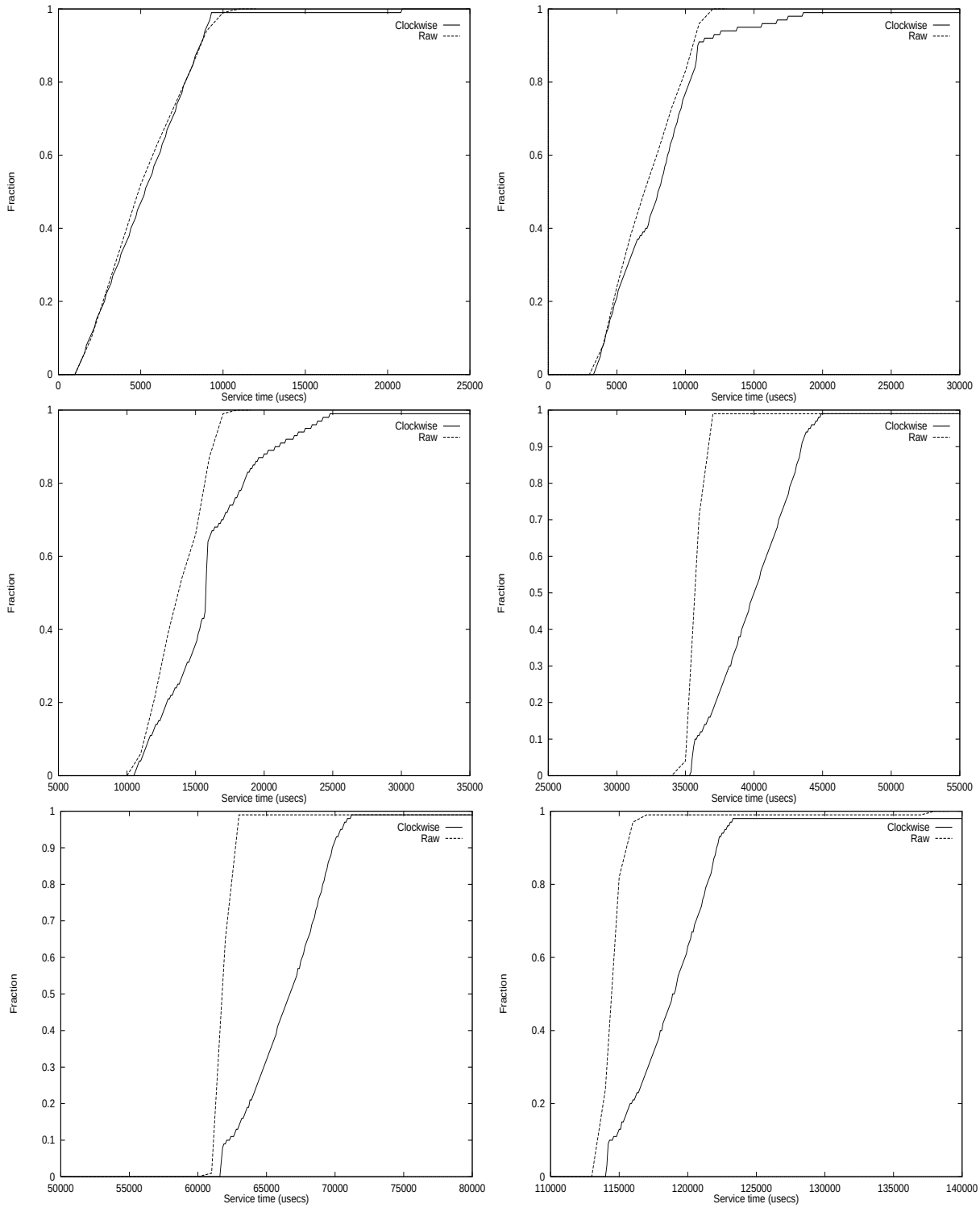

Figure 7.3: Read performance in a dynamic partition on zone 0 of a Quantum Atlas-II disk. Upper left represents the performance on for transferring a block of 1 KB, followed by 16 KB, 64 KB, 256 KB, 512 KB and bottom right presents the time to read 1 MB.

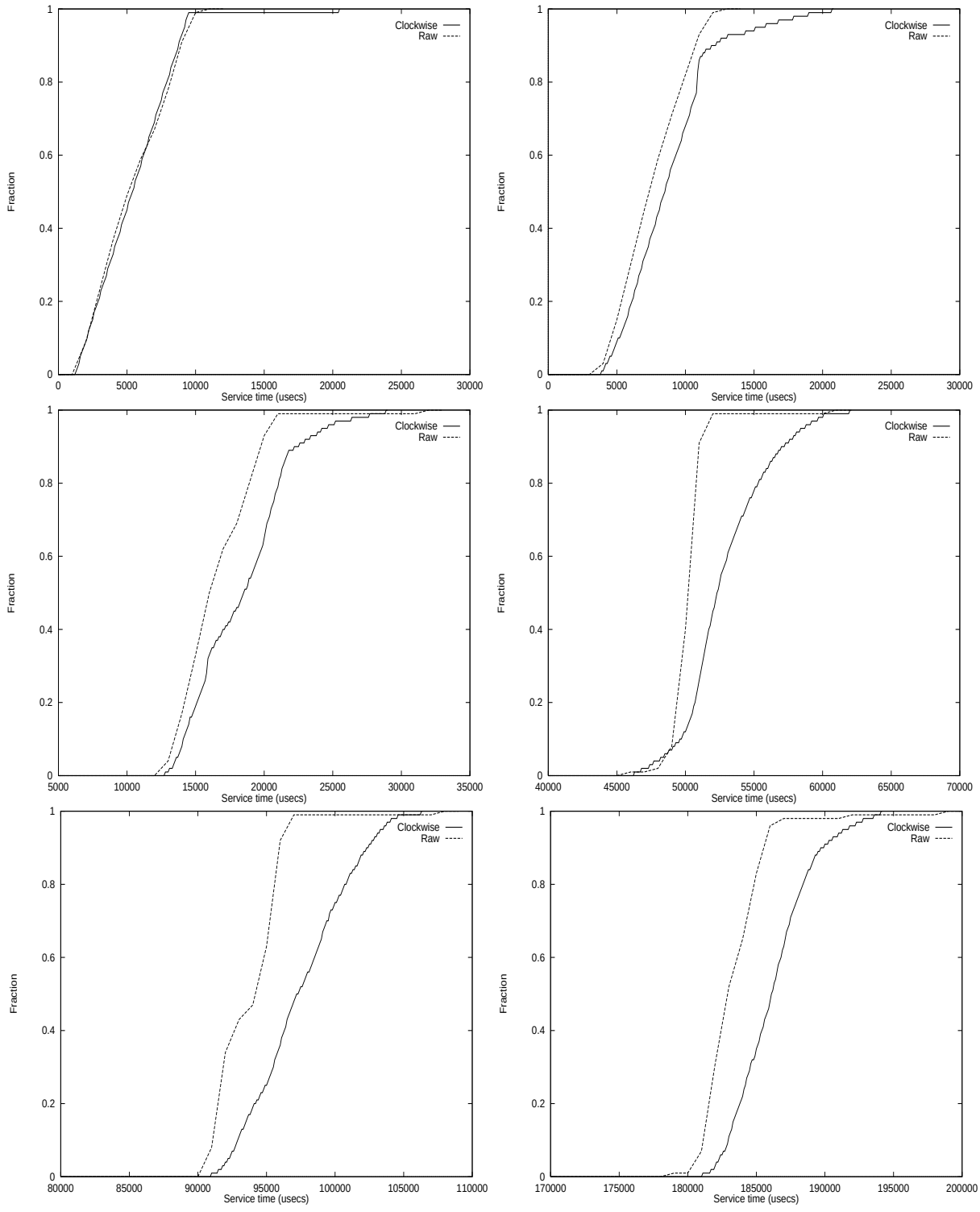

Figure 7.4: Read performance in a dynamic partition on zone 11 of a Quantum Atlas-II disk. Upper left represents the performance on for transferring a block of 1 KB, followed by 16 KB, 64 KB, 256 KB, 512 KB and bottom right presents the time to read 1 MB.

aware of which blocks have what size. If Clockwise decides to move dynamic partition blocks through the array, the help of the storage application is required for the re-organization. This can be accomplished by typing the stored data and integrating user functions in Clockwise for re-organization purposes. However, this implies a tremendous complication of the storage system: currently the stored data is an untyped array of disk sectors without user application involvement.

7.2.2 Clockwise disk caches

The problem of unaligned dynamic partition blocks is traditionally solved by disk caches. These hardware disk caches store the last portion of a track and if the next request that arrives in the disk controller is a sequential one, the remainder of the last track is delivered from the disk cache instead of from disk. The problem with this approach is that current disk caches are relatively small. The Quantum Atlas-II disk, for example, only uses a 512 KB disk cache and when the disk is used to deliver a few continuous-media streams, each of which uses block sizes that are equal to or larger than the size of a track, the contents of the disk cache is wiped quickly.

To optimize for sequentially accessed dynamic partitions, Clockwise implements its own disk cache in main memory. This disk cache holds the remainders of tracks of which one part is read by a storage application and the other part is not yet read because it is part of the next dynamic partition block. The assumption is that most (continuous-media) dynamic partitions are read sequentially, so by pre-reading the remainder of a track into the Clockwise disk cache, a sequential next request simply copies the data from the Clockwise disk cache. The Clockwise disk cache is implemented as an array of track buffers that are replaced on an LRU basis. Clockwise storage applications can flag that they are planning to transfer data sequentially.

A hardware disk cache usually only implements read-ahead when there is no other disk activity. Given that all of the real-time requests are executed in a burst by the ΔL scheduler, it is unlikely that the disk's read-ahead policy has any effect: a request from another application is queued in the disk the moment the disk has transmitted the last part of the previous requests to the host. It means that many read-ahead operations will be interrupted and disks do not finish the read-ahead operations.

Clockwise can apply a smarter policy because it has knowledge of all the task service times, and it knows for certain whether or not a dynamic partition is accessed sequentially. When there is enough time available, either in the predicted service time or in ΔL slack time, to transfer the remainder of a track across the SCSI bus into the Clockwise disk cache, Clockwise rounds up the requested number of sectors so that the request ends on a whole track boundary. The reason that there only needs to be SCSI-bus time available is because the service-time prediction has already included the time to perform an entire extra rotation. Only when a start sector of a request is not aligned to a track boundary, the request's first sectors are not already read into the Clockwise disk cache, and the remainder of a track needs to be read from caching purposes, a time estimate needs to be made to read the remainder of the track (rotational time and SCSI-bus time). Again, if there is slack time available or the transfer takes less time than is anticipated for, the remainder can also be read into the cache.

Consider for example, the transfer of a dynamic partition block with a size of 1 MB, or 11 tracks and 79 sectors, on the outer zone of a Quantum Atlas-II disk. If block 0 of the dynamic

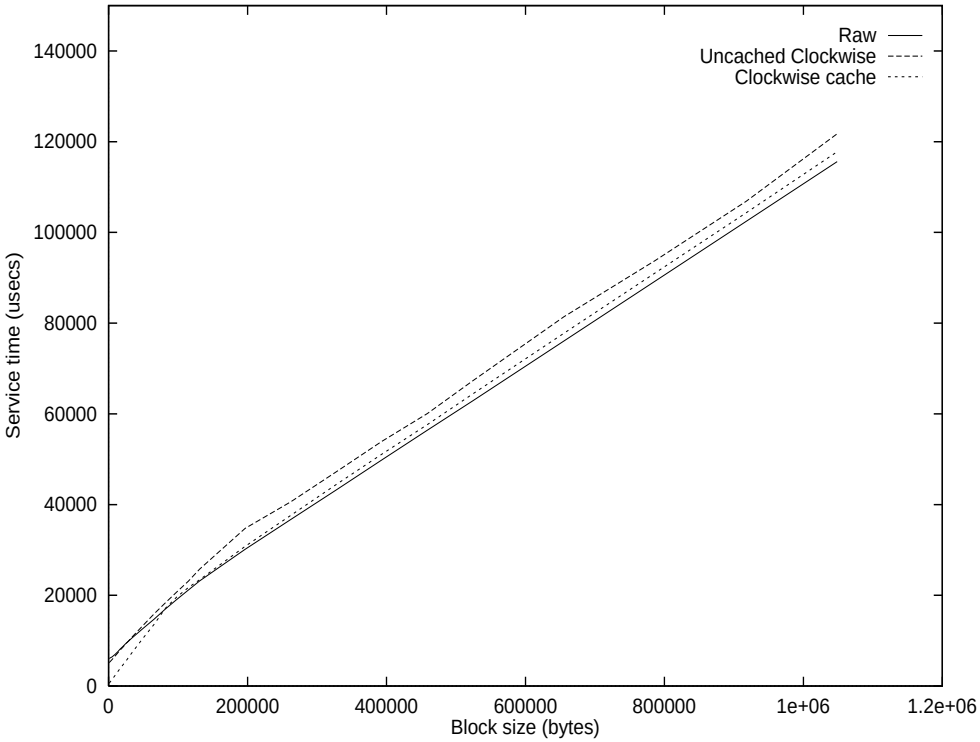

Figure 7.5: Mean service times for raw performance, uncached (*i.e.*, no read-ahead) and Clockwise cached disk requests of blocks in a dynamic partition on zone 0 of a Quantum Atlas-II disk.

partition starts at disk address 17, a request for 2048 sectors from the client is translated into a disk request of 2131 sectors.[2] The remainder of 83 sectors are stored in the Clockwise disk cache. When dynamic partition block 1 is requested by the application, 83 sectors are first retrieved from the disk cache followed by a request for 1969 sectors: precisely 11 tracks. The price for this optimization is that in other cases 12 tracks need to be read.

Figure 7.5 shows the mean service times of requests with varying block sizes in the 'raw' case, the 'uncached' case (*i.e.*, both disk read-ahead policy and Clockwise cache are disabled) and with the Clockwise disk cache enabled. Request service times are shortest in the raw performance experiment. When both read-ahead and Clockwise disk cache are disabled, the requests complete on average slightly more than half a rotational delay later because, on average, the disk has to wait half a track for the requested start sector of a dynamic partition block.

The reason the service times are slightly higher than half a rotational delay is because the dynamic partition contains bad and error blocks. When the disk tries to read a bad block, it retrieves a replacement block instead, when the disk tries to read a block with an error, it retries on the block. In both cases, the actual service times deteriorates. The raw performance measurement does not have this problem because these measurements are executed on an error free part of the disk.

The Clockwise disk cache experiment shows that the service times can be close to what is

[2]The Clockwise superblock is 1 sector long and starts at sector 16.

measured in the raw performance experiment. Actually, it is expected that the mean service time for requests with the Clockwise disk cache enabled is identical to the raw mean performance because the same amount of information is read. However, the bad and error blocks alter the mean in Figure 7.5 by 1–2 ms, which is the measured difference between service times of both experiments.

The figure also shows that for short transfers, *i.e.*, shorter than a track, the Clockwise disk cache is beneficial: many requests hit in the cache and do not go to disk at all. In fact, sequential 1 KB transfers show Clockwise cache hit rates of almost 99 %. The reason that the mean service time with the Clockwise disk cache enabled is lower than for what is measured in the raw performance experiment, is because in 99 % of the cases, requests do not have to contact the disk at all, thereby saving on the SCSI overhead and on the rotational overhead.

Figure 7.6 shows the cumulative distribution of 512 KB transfers from a dynamic partition on the outer zone of a Quantum Atlas-II disk. The figure shows several performance measurements. The curve labelled 'raw performance' are the raw service times, as described in Chapter 4, for reading a 512 KB block. The 'uncached performance' curve shows the performance implications when disk read-ahead is disabled and the Clockwise disk cache is not used.[3]

The experiment that is labelled '1-track read-ahead' in Figure 7.6 shows the performance implications of the Clockwise disk cache. When transferring 512 KB blocks from a dynamic partition that is laid out on a 179 sectors per track zone, 30 % of the requests need to read 5 tracks, and 70 % of the requests need to read 6 tracks as is shown in the figure. The 5-track transfers complete much quicker than the raw performance and the 6 track transfers complete precisely one rotation later than the 5-track transfers. The curve labelled 'next track' represents the raw performance for transferring 6 whole tracks. The reason why the '1-track read-ahead' performance is not precisely identical to the performance of the 'next track' curve when reading 6 tracks is caused by the current (naive) Clockwise disk cache implementation. The curve for the '1-track read-ahead' also shows that there is quite a long tail in the distribution. The tail is caused by bad and error blocks.

If dynamic partitions are not stored sequentially the disk cache is of no use. However, most Clockwise dynamic partitions are usually created atomically and are stored consecutively on a disk. Also, dynamic partitions can be re-organized so that they are stored consecutively. Finally, to playback earlier recorded continuous-media data from dynamic partitions, the dynamic partitions are usually read sequentially.

7.2.3 Write performance

As is already shown in Section 4.3, the Quantum Atlas-II disk has an odd write performance. As an example, Figure 7.7 shows the cumulative distribution of the raw transfer of 10 tracks, 10 tracks plus a sector and 11 tracks to the outer zone of a Quantum Atlas-II disk. When 10 or

[3]While taking the measurements an 'Amazing Discovery' is that by enabling the Quantum Atlas-II disk controller's cache, the performance degrades by an extra 10 ms per request. By measuring the SCSI $\overline{\text{BSY}}$ signal it appeared as if the disk is trying to coalesce future SCSI request with the current request, or that is is busy performing a prefetch. In any case, the disk is doing something it is not supposed to do. Hence, to use these disks efficiently, any sensible system needs to switch off the disk cache and implement their own cache.

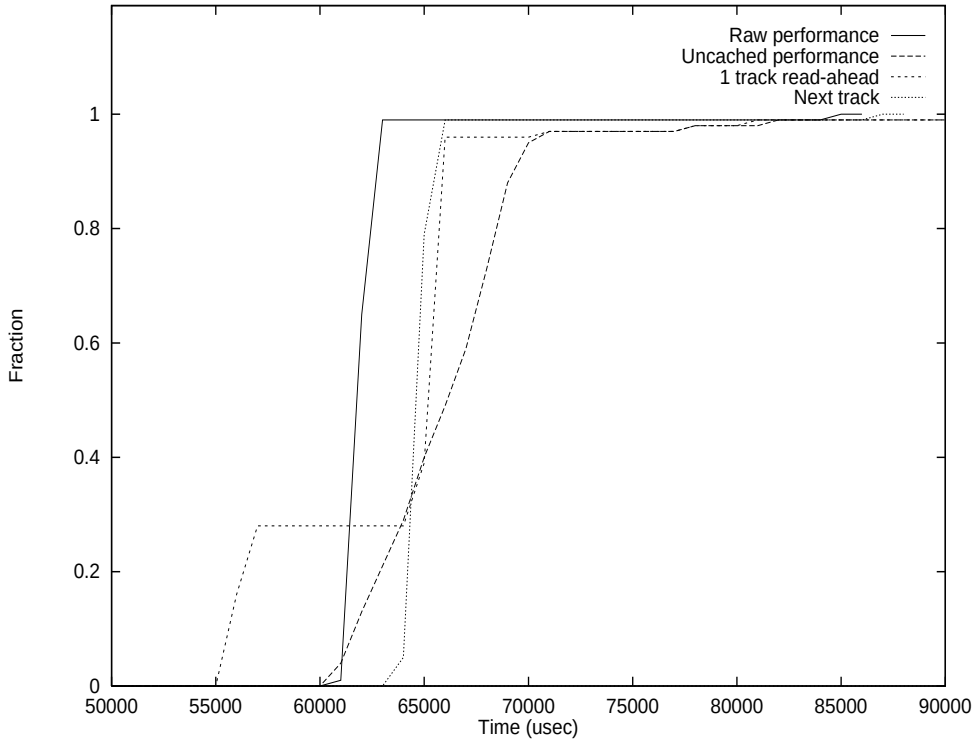

Figure 7.6: Clockwise dynamic partition performance with cache enabled for 512 KB transfers.

11 tracks are transferred, the distribution follows the timings that can be expected for 11 or 12 rotations, respectively. If, however, an extra sector is added to a whole number of tracks transfer, the entire cumulative distribution is shifted by the time to almost perform an entire extra rotation. One would expect that when streaming is implemented well, the shift would only be marginal. This situation is not different in other zones of the Quantum Atlas-II.

In the light of the odd write behavior of the Quantum Atlas-II disk, it is difficult to measure the write service times. As described earlier, Clockwise does not align dynamic partition blocks to track boundaries, which means that some transfers are performed in an efficient manner, while in other cases the disk needs to perform extra rotations to write the data to disk.

To learn of the maximum write service times for the Quantum Atlas-II disks, a number of write experiments with a varying block size is performed on the inner and outer zone of the Quantum Atlas-II disk. Although a number of block sizes is measured, only the transfer of a single representative block size is looked at more closely. Figure 7.8 shows the cumulative distribution of the transfer times for a megabyte block to zone 0 (top) and zone 11 (bottom). The figure shows four curves: the raw and Clockwise service times when the disk is connected through a NCR53c810 Fast SCSI-2 controller and the raw and Clockwise performance when the disk is connected by a NCR53c875 Ultra-Wide SCSI controller.

When the disk is connected by a NCR53c810 SCSI controller, most raw write service times for zone 0 are between 111 and 120 ms. If, on the other hand, NCR53c875 Ultra-Wide SCSI controllers are used to transfer the data to disk, most raw service times reduce to what can be

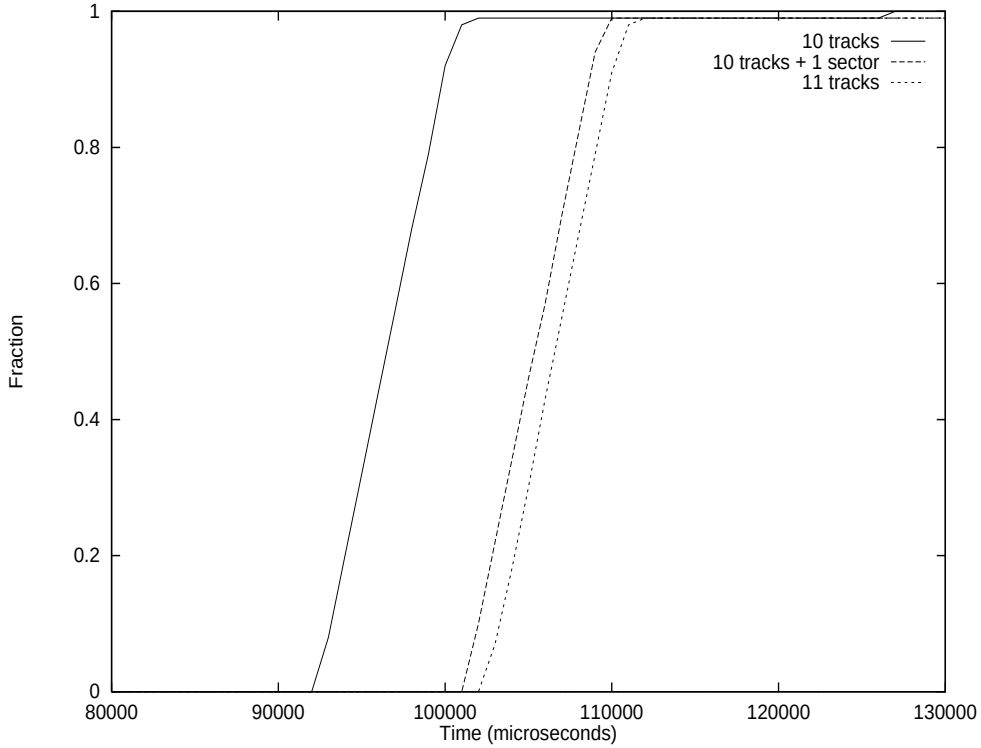

Figure 7.7: Cumulative distribution of a 10 track, 10 track plus a single sector and an 11 track raw transfer in zone 0 of a Quantum Atlas-II.

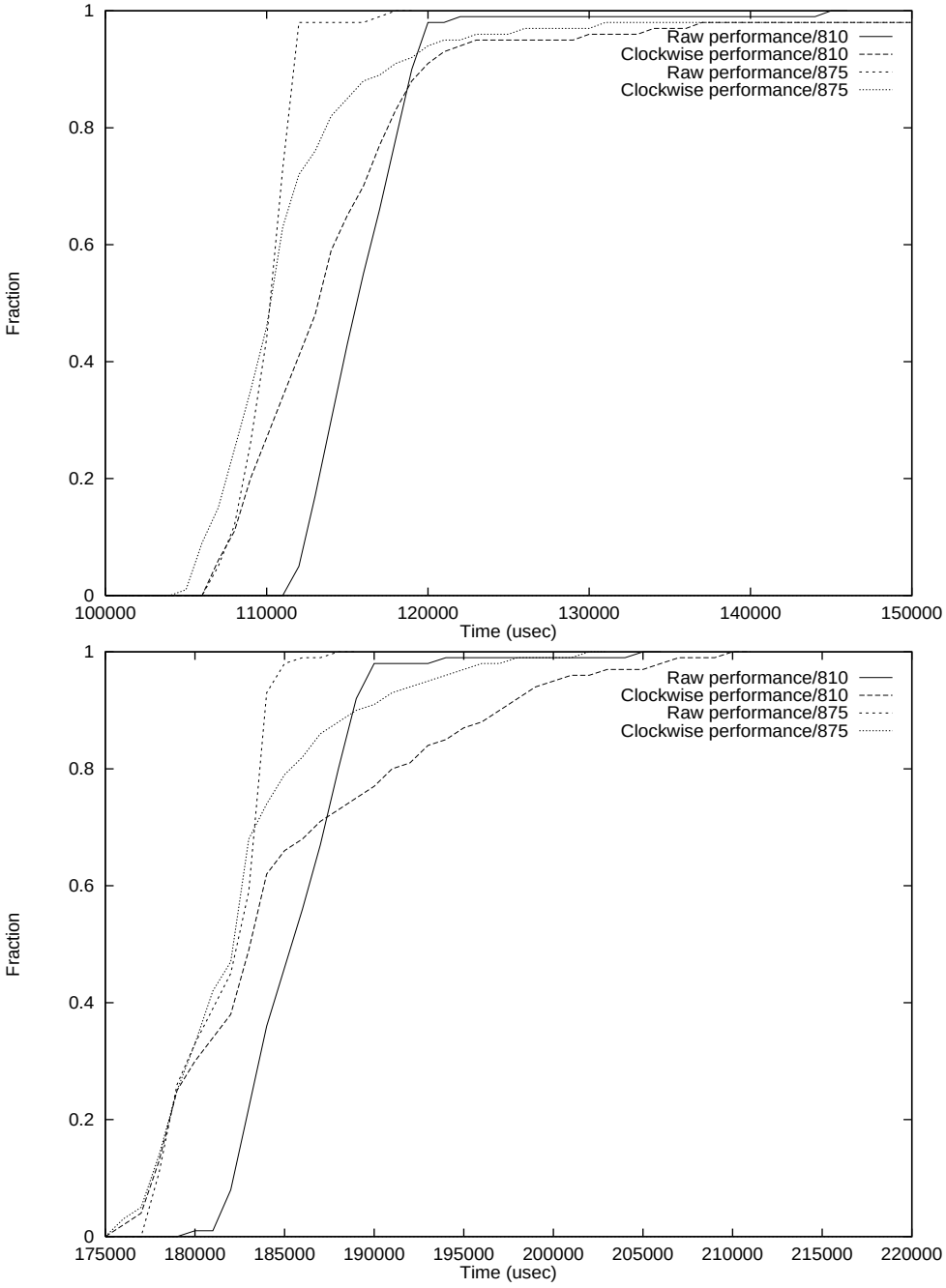

Figure 7.8: Cumulative distribution for writing a block of 1 MB to zone 0 and zone 11 of a Quantum Atlas-II disk.

expected by the physical parameters of the disk. The Ultra-Wide SCSI controller uses a SCSI-bus speed that is twice as high when compared to the Fast SCSI-2 controller. This higher bus speed is used by the Quantum Atlas-II disk controller to transfer data at a higher speed across the SCSI bus.[4] Since the raw service times decrease in both zones with the NCR53c875, it seems the Quantum Atlas-II's disk is not really a Fast SCSI-2 disk, but one that needs to operate at higher bus-clock speeds.

Also, the raw performance measurements with the Ultra-Wide SCSI controller for zone 0 shows a tail in the service-time distribution, which is not caused by bad blocks. This implies that the disk misses a rotation because it does not fetch the data quickly enough from host memory. Clearly, there is something wrong with the Quantum Atlas-II disk write functionality. Unfortunately, the only way to find out what the disk is really doing on (large) writes, is by analyzing the internals of the disk's firmware. This can only be performed with the help of the manufacturer of the disks.

When Clockwise is used to write data to disk, the distribution of the service times changes so that some requests complete quicker than the aligned (raw) requests and some requests complete slower than aligned requests for both the NCR53c810 and NCR53c875 SCSI interfaces. It is unclear why some disk requests complete quicker than the aligned disk requests. Again, the manufacturer's help is required.

Most of the disk requests that take longer to complete, require an extra rotational delay, just like reading data from unaligned dynamic partition blocks. Therefore, using service-time predictions that are one rotational time longer than the raw performance numbers is required. However, given that a few requests take even longer to complete, it seems worthwhile to include an extra safety margin in the service-time prediction.

In any case, the Quantum Atlas-II disk does not seem a good candidate for a real-time file system: it is too unpredictable in its write behavior.

7.2.4 Multi Quantum Atlas-II service times

Clockwise can layout dynamic partitions across multiple disks. Instead of reading and writing a single disk at a time, a storage application can transfer data to a number of disks at the same time thereby increasing the maximum available disk bandwidth.

The performance of a set of parallel Quantum Atlas-II disks is measured by an applications that issues a number of read requests of varying sizes to a dynamic partition that is laid out on three Quantum Atlas-II disks. The dynamic partition is laid out such that if block n is located on disk d, block $n + 1$ is located on disk $(d + 1)mod3$.

Figure 7.9 shows the performance speedup for reading data from two or three disks simultaneously compared to the read performance of a single disk. It shows that two disks are 1.98 as fast as a single disk and three disks are approximately 2.95 as fast as a single disk. The reason for not being precisely 2 and 3 times as fast is for the same reasons as is the case for the raw multi-disk performance measurements that are described in Chapter 4. The reason for not showing the

[4] Again an 'Amazing Discovery': the Quantum Atlas-II disks that are used for the experiment are Fast SCSI-2 disks and hence, should not work with an Ultra-Wide SCSI bus.

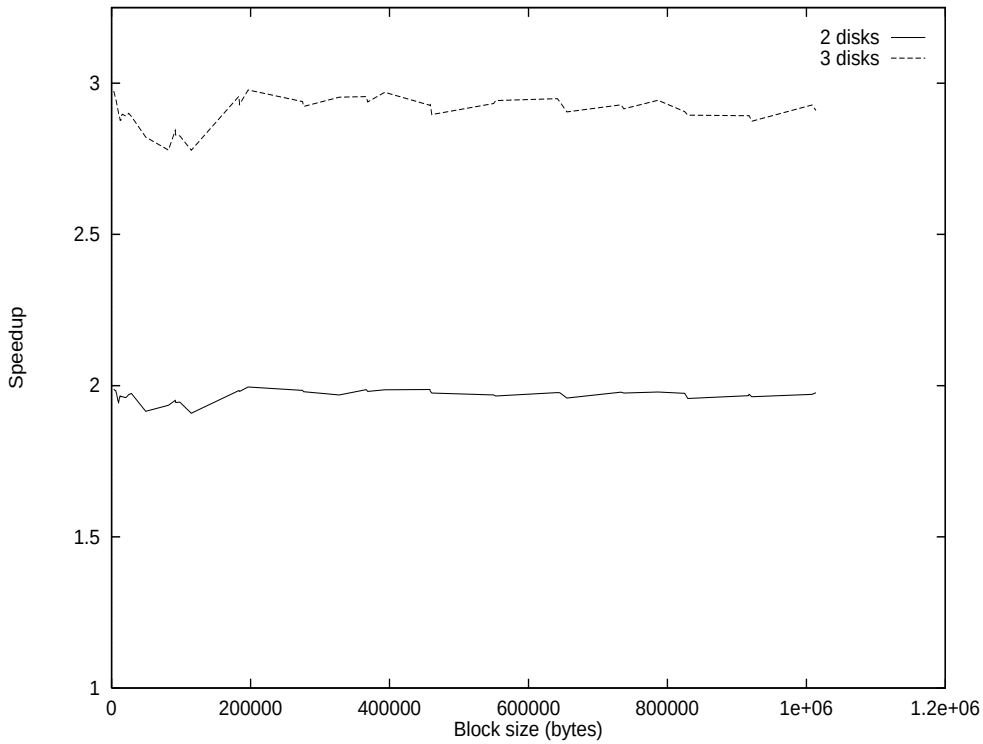

Figure 7.9: Clockwise multi-disk speedup.

actual service time speed-up is because there are only minute performance differences between service times that do not show up in a figure.

Given that the speedup figures closely resemble those of the raw performance measurements, it can be concluded that when up to three parallel Quantum Atlas-II disks that do not share a SCSI bus, transfer data simultaneously, the service times are almost equivalent to the service time of a single disk. Hence, the service-time predictor does not really have to worry about interferences between multiple disks.

The service-time predictor can base service-times predictions for multiple disks that are written to, also through the earlier described method. The reason for this is that as is shown in Figure 4.14, three parallel disks are almost three times as fast as a single disk, just like for the disk read experiment.

When the Quantum Atlas-II disks are serviced by NCR53c810 Fast SCSI-2 controllers, and the PCI bus is used heavily for other traffic, the PCI bus can become a bottleneck. In this case the actual service times of requests that are executed on the Quantum Atlas-II disks can deteriorate. This is a fundamental problem and is discussed in Chapter 8 in more detail.

7.2.5 Multi Seagate-Cheetah service times

Three 4.5 GB Quantum Atlas-II disks have a total storage capacity of 13.5 GB and can deliver or receive data at a maximum rate of approximately the speed of a 155 Mb/s ATM network. As is

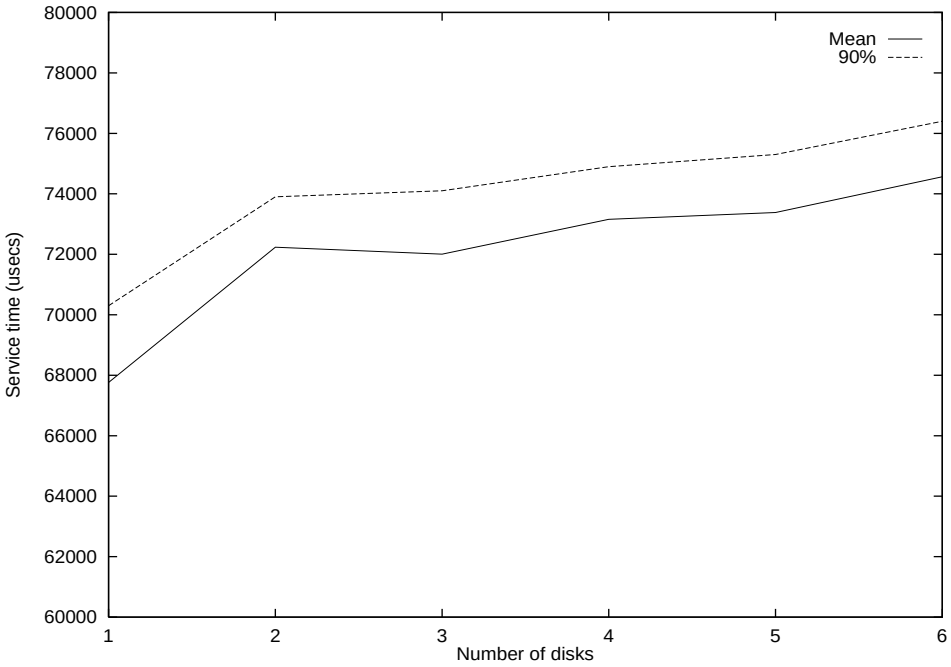

Figure 7.10: Clockwise read service times for $1 \ldots 6$ Seagate Cheetah disks when transferring 1 MB blocks.

shown in Section 4.3, the raw performance of a Seagate Cheetah is a factor of 1.5 higher than a Quantum Atlas-II disk and a combination of Seagate Cheetahs with Ultra-Wide SCSI controllers can cope with higher network speeds.

To measure the Seagate Cheetah performance under Clockwise, a number of these disks is connected to a Clockwise machine and Clockwise is prepared to schedule Seagate Cheetah disks.[5] As is shown in Figure 4.17, the performance of a set of parallel Seagate Cheetah disks is determined by the way they are connected to and used by Clockwise. When more than two disks are connected to the same Ultra-Wide SCSI bus and are used at the same time, the utilization per disk on a shared string quickly decreases. Also, when more than six disks on three SCSI buses are used the aggregate performance of the disk may degrade because of memory-saturation problems.

To avoid overload problems, the Clockwise server is configured with only three NCR53c875 Ultra-Wide SCSI cards, and on each of these buses only two Seagate Cheetahs are used at the same time. These six disks are used for sequential throughput measurements. The goal of the measurements is to learn if the Clockwise scheduler can sustain high data rates.

Figure 7.10 shows the service times for reading megabyte blocks from 1, 2, ... 6 disks. The figure shows that there is almost a constant service time between 2 and 5 disks. For 2 to 5 disks, the service time for a megabyte block transfer is between 72.0 and 73.3 ms for the average case and between 73.9 and 75.3 ms for the 90-percentile of the requests. The difference of 4 ms between 1 and 2 disks is caused by the sharing of the SCSI bus. Since the minimum transfer size

[5] My gratitude goes to Frans Jonkers for measuring the raw performance of a Seagate Cheetah disk.

across the SCSI bus is approximately 170 KB, the minimum SCSI-burst time is 4.3 ms. When 6 disks are used at the same time, the service time degrades by 1.2 ms. However, when 6 disks are used in parallel and the 90-percentile of the service times is used, Clockwise can be used to transfer 78.5 MB/s for sequential requests without missing deadlines.

Write service times have not been measured, but it is expected that the service times for writing to 6 parallel disks has the same service-time implications as for reading 6 parallel disks. The reason for this is that Figure 4.17 does not show a performance difference between disk reads and disk writes when seven or less disks are used in parallel.

7.2.6 Bad and error blocks

When a disk controller finds out that a requested sector cannot be read or written without an error, it replaces the *bad sector* with a replacement sector. These replacement sectors are located at different locations on disk, and when a client reads or writes a bad sector, the disk controller reads or writes the replacement sector instead of the originally requested sector.

It usually takes longer to read or write a bad sector because of the disk's replacement policy. The disk heads need to be relocated to the replacement sector, the replacement sector needs to be read or written, and the disk needs to return to the original location to continue reading and writing. Unfortunately, most disk vendors do not publish what a disk does when it encounters a bad block. Detailed measurements are required to reverse engineer the bad block replacement policy.

A second type of disk errors are so-called retry errors. Certain blocks are not read correctly in one try, and one or more retries are required to retrieve the information from disk. To retry, the disk waits for a full rotation to the error block and reads the data again.[6] Bad sectors and error blocks are further referred to as bad blocks.

The Clockwise disk scheduler predicts service times through the raw-performance measurements, which are described in Chapter 4. While performing these raw-performance experiments, no bad blocks are found, so when a bad block is found during a Clockwise transfer, the predicted service time may be a service time that is too short. When Clockwise does not consider bad blocks, and a dynamic partition is stored on the bad block, reading the data through a real-time application may lead to deadline misses: transferring the block takes more time than is originally anticipated for.

There are several ways to deal with bad blocks. The simplest solution is that Clockwise does not consider bad blocks at all, and accepts an occasional deadline miss. Unfortunately, when a disk gets older and more blocks become bad, Clockwise will suffer from more deadline misses. Another solution is to anticipate on the bad blocks, and increase the service-time prediction to include the time to perform the replacement operation or to perform one or more retries. Given that the replacement or retry operation can take a considerable time, such a strategy reduces the schedulability of the system tremendously.

[6] The Quantum Atlas-II disk does not keep a history of retry blocks, even though they are always the same sectors. In fact, another 'Amazing Discovery' is that it is not possible to inform the Quantum Atlas-II disk to ignore or report errors on retry blocks rather than just retrying.

Another solution to the bad-blocks problem is not to use dynamic partitions with bad blocks for real-time transfers. When a block becomes bad and it is used inside a real-time dynamic partition, Clockwise copies the data from the dynamic partition block to a free dynamic partition block that has not bad blocks. The bad dynamic partition block can then only be used for best-effort data storage.

Since best-effort traffic is also explicitly scheduled, Clockwise needs to be able to predict the service time for a dynamic partition with a bad block. Again, there are two possibilities. Either the bad blocks are used for storage and the disk's replacement policy is used to find the replacement blocks, which complicates service-time predictions and reduces the schedulability of best-effort traffic in ΔL time. Alternatively, the disk is instructed not to retry or re-map on errors, but to report this to Clockwise. Clockwise can, in that case, adjust the predicted service time for the request and re-issue the request for the bad block. Unfortunately, the latter option cannot work for the Quantum Atlas-II disk because this disk seems to ignore retry hints.

For now, Clockwise simply uses the disk's replacement policy to re-map or retry the bad block in a best-effort dynamic partition. To find out which blocks are bad, Clockwise periodically asks the disk controller which blocks are bad and it keeps a history of blocks that required retries. When a storage application requests a bad block, Clockwise adjusts the service-time prediction for the best-effort dynamic partition block.

7.3 QoS crosstalk prevention

One of Clockwise's design goals is to prevent QoS crosstalk. In Section 6.5.5 an overview is given of Clockwise's approach to prevent QoS crosstalk from happening and simulation results are presented that show that the ΔL scheduler can guarantee service even when there are misbehaving applications.

To measure the influence of misbehaving applications on behaving applications, a Clockwise load generator and a measurement application are run simultaneously. The load generator generated a disk load that is higher than it has requested by means of its QoS parameters. The measurement application is started such that the disks have no slack time available ($\Delta L = 0$). The measurement application issues a load within its QoS contract.

As expected, both the LST and EDF scheduler start missing deadlines in the behaving measurement application as soon as the load generator generated a load that is higher than the requested rate. The reason for this is that these schedulers declare the system idle when there are no real-time requests queued. Both schedulers start executing requests from the load generator application as soon as an idle period is found without considering that the measurement application can release a real-time job at the next instant. Since the ΔL scheduler does consider the case that the measurement application can release a request at the next instant, this scheduler correctly never executes a load generator request before its release time. Hence, the measurement application never misses a deadline.

7.4 Combined network and disk performance

When Clockwise is used to playback or record AVA continuous-media data, it is important to understand the performance implications of the scatter-gather functionality of Clockwise. Especially when a Motion-JPEG compressed AVA video stream is recorded or played-back, the scatter-gather lists that are presented to Clockwise can be a long list of small buffers. The 'Wallace and Grommit: A Close Shave' movie, for example, has an average AAL5 packet size of 280 bytes, the 'Hunt for Red October' movie has an average of 420 bytes per AAL5 packet.

The performance implications of the scatter-gather capability of Clockwise are twofold: Clockwise needs to verify each of the presented buffer pointers before inserting the addresses into the SCSI chip-sets and PCI performance can be less efficient for transferring such small quantities of information. Given the AAL5 packet sizes, Clockwise needs to verify between 2,000 and 4,000 addresses for megabyte dynamic partition blocks per block. When Clockwise schedules at its maximum rate of 80 MB/s, only 3 % of the available CPU capacity is used to validate the buffer pointers.

The performance implications of sending small (non-aligned) buffers across the PCI bus can be analyzed by a PCI bus simulator, which uses the timing information from Section 4.3. Figure 7.11 shows the performance implications of receiving or sending small AAL5 packets between the OPPO and NCR controllers through main memory. In the figure, four situations are simulated: recording and playback of data with either a NCR53c810a or NCR53c875 SCSI controller. Because of the PCI overhead, the device that performs the memory-read operation implements the bottleneck operation. The NCR53c810 uses a 64-byte FIFO, the NCR53c875 uses a 512-byte FIFO and the OPPO uses a 256-byte transfer buffer.

As expected, the best utilization is achieved when data is recorded through NCR53c875 SCSI controllers, the worst-case utilization is reached when data is recorded through the NCR53c810 SCSI controllers. The reason that the NCR53c875 combination shows the best performance results is because the overhead for performing memory-read operations is divided over a large number of transferred bytes. When only a few bytes are transferred from memory to PCI card, the overhead accounts for a relatively larger portion of the total transfer time. The minimal combined performance is approximately 43 MB/s, the maximal combined performance is approximately 57 MB/s. Since the currently available network runs at 155 Mb/s, Clockwise easily overloads the network capacity.

The zig-zags in the figure are caused by non-aligned AAL5 packets. If the packets that are transferred are slightly larger than the size of a hardware FIFO, an extra PCI transaction is required to obtain the last few bytes from memory. For example, when all network packets are of size 336 bytes and data is recorded on a disk that is connected by an NCR53c810a SCSI controller, three large PCI transactions and one small PCI transaction are required to transfer the packet to disk. In this case the PCI utilization is only 33.6 %. Also note that memory saturation problems are not taken into account for the simulations: the actual maximum performance is likely to be lower.

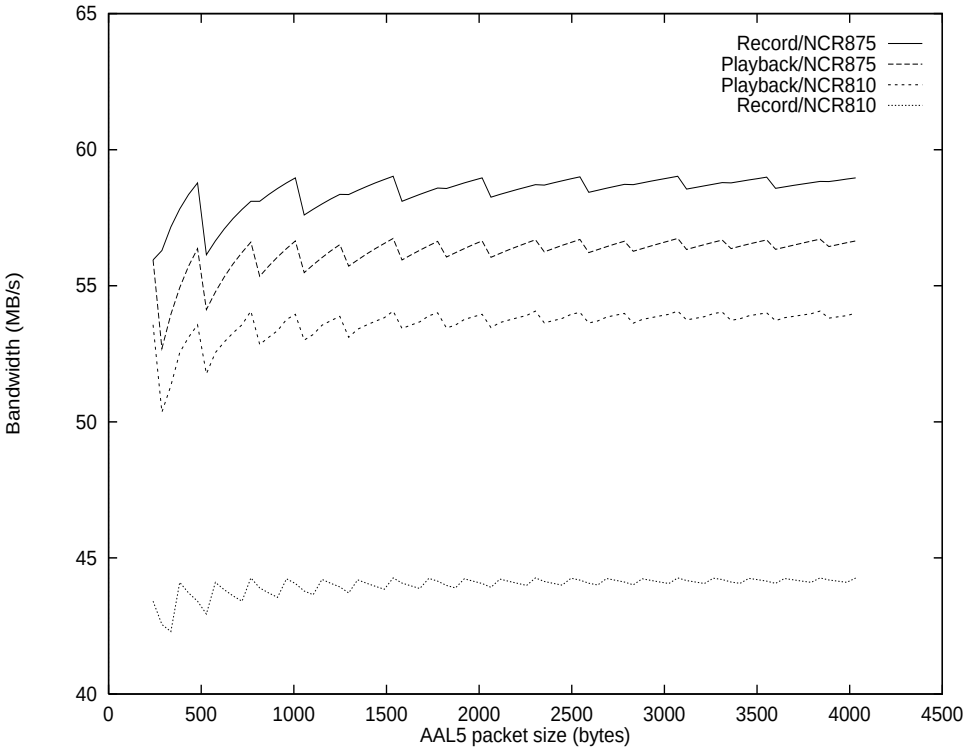

Figure 7.11: Combined OPPO and NCR throughput.

7.5 CPU usage

While performing the multi-Seagate Cheetah performance experiments, CPU measurements are conducted to learn how much CPU time is involved in running Clockwise when megabyte buffers are transferred. Statistics are gathered every time the underlying operating system, Nemesis, performs a context switch and the program counter is saved every time the clock ticked [14]. Since Clockwise is currently embedded in the NCR SCSI driver the total CPU cost for running the storage system are obtained.

Table 7.2 shows the load on the CPU for the Clockwise server when $1 \ldots 6$ disks are used in parallel. Clearly, the table shows that driving all of the SCSI buses at the same time and executing

Number of disks	CPU load
1	0.008
2	0.014
3	0.020
4	0.028
5	0.036
6	0.041

Table 7.2: CPU load for driving $1 \ldots 6$ disks

the Clockwise server does not involve much CPU processing.

The reason for the low CPU load is that the used SCSI cards, the NCR family of controllers, perform most of the SCSI handling by themselves. The CPU is only required to insert new parameters into the SCSI chips, to the start the chips, to process the final interrupt and to signal the user application of the finished operation. Clockwise performs dynamic partition disk-address translation and implements an EDF queue to schedule requests. So, when a measurement application sends requests back-to-back to Clockwise with an aggregate bandwidth of approximately 80 MB/s, only 4 % of the CPU is required to execute Clockwise.

Obviously, when smaller transfer buffers are used, the CPU load is increased as well. The Seagate Cheetahs transfer 16 KB of data in approximately 7.9 ms. Since the amount of work that needs to be performed in Clockwise to transfer a small buffer is the same as for transferring a large buffer, running six Seagate Cheetahs for 16 KB transfers costs approximately 40 % of the CPU.

7.6 Summary

Chapter 6 shows that the ΔL scheduler is good in scheduling a combined best-effort and real-time load on the same set of disks. This chapter first shows that the performance results from the earlier chapter are also valid in a real system. The simulations are validated by showing that measured and simulated performance results are (almost) identical.

The service-time predictions in Clockwise are based on the raw-performance measurements that are described in Chapter 4. In this chapter it is shown that the time to perform an extra rotation needs to be added to the predicted service times. The reason for this is that Clockwise dynamic partition blocks are not aligned to track boundaries, and in a worst-case scenario, the disk needs to wait almost a full rotation before it can commence a transfer. Also, it is shown that a Clockwise disk cache can be beneficial in reducing the actual service time for dynamic partitions that are read sequentially. The write service times are much harder to predict: it seems that the Quantum Atlas-II disk is not good in writing large blocks. Only when Ultra-Wide SCSI controllers are used for the Fast SCSI-2 disks, the service times are reasonably predictable. In any case, it is advantageous to include a safety margin in the write service time predictions.

This chapter also shows that when dynamic partitions are stored on multiple disks, the predicted service times on concurrently accessed Quantum Atlas-II or Seagate Cheetah disks can be based on the service time of a single disk. When a SCSI bus is shared between disks, the predicted service times need to include the bus-transfer time of a disk cache line size worth of data. The service times for up to three parallel Quantum Atlas-II disks and to a limited extend, up to six parallel Seagate Cheetah disks are presented.

Lastly, this chapter presents a few loose ends that are related to the actual performance of a Clockwise system. It is shown that Clockwise prevents QoS crosstalk by strictly enforcing an EDF queue and executing requests that arrive too quickly in slack time. Further it is projected that theoretical combined performance of networks and disks is maximally 58 MB/s on a Pentium-Pro based machine. It is shown that executing Clockwise is not CPU consuming: when six Seagate Cheetahs are used to transfer data at approximately 80 MB/s, only 4 % of the CPU is required to

transfer megabyte blocks in non-scatter-gather mode.

Chapter 8

Discussion

Clockwise is now a single server and stand-alone file system that is capable of combining real-time data requests with best-effort UNIX file requests. Clients typically contact Clockwise through an interface such as NFS to retrieve and store their best-effort data and use the Clockwise session manager to record or playback continuous-media files.

There are a number of ways in which the Clockwise work can be extended. Currently, only Quantum Atlas-II and (in a limited way) Seagate Cheetah disks are schedulable by Clockwise. The Quantum Atlas-II disks are connected by a single SCSI bus to the host machine. The total storage capacity of the system is only 13.5 GB. Such numbers are not realistic for a real mixed-media storage system.

The Seagate Cheetah configuration that is measured in Chapter 4 and Chapter 7 is a more realistic configuration. Each of the four SCSI buses can host a number of Seagate Cheetahs and the disks that share a SCSI string can easily overload the SCSI bus. When all Seagate Cheetahs are transferring data simultaneously, they can easily overload the PCI bus and memory system of a standard PC. A true high-bandwidth mixed-media storage system needs a method of scheduling SCSI and PCI buses and memory or else service times are unpredictable.

A modern PC is capable of sustaining 100 MB/s from disk. When network transfers are also taken into account, this number is reduced, in theory, to approximately 59 MB/s for recording and 56 MB/s for play backs. Although this is a high throughput, there exist applications that require more than this amount of throughput.

The only way to enlarge the sustained bandwidth of the current Clockwise configuration is by using multiple Clockwise servers in parallel. Although faster PCI buses may extend the maximum performance, there always exist applications that require more throughput. Also, from an economical and fault-tolerance perspective, it seems wise to use parallel (cheap) commodity hardware instead of using faster machinery sequentially. VxFS is an example of the latter approach, with a measured throughput of almost 1 GB/s [98].

Clockwise currently runs on Nemesis, a novel operating-system kernel, that is especially designed to execute continuous-media applications. Although Nemesis is suitable for all kinds of continuous-media application experiments, it is still a prototype operating system. This means that the operating system is not yet stable enough to host a departmental file server. To be able to use Clockwise for departmental file storage, an effort is underway to port Clockwise scheduling

techniques to existing operating systems.

As is shown in Section 7.2, Clockwise requires service-times predictions for the scheduler's admission test and for online service-time predictions to allow best-effort tasks to take precedence over real-time tasks. The current Clockwise prototype uses the raw performance measurements from Chapter 4 to calculate these values. However, the process of measuring all of these values and to understand what a disk is doing, is usually a tedious and time consuming task. A method is required to obtain such numbers in a semi-automatic manner.

The amount of secondary storage capacity is always limited: disks are expensive. For Clockwise it is viable to support tertiary storage devices for storing continuous-media data and for backup purposes.

The remainder of this chapter describes the extensions in more detail.

8.1 Parallel disks

Digital audio and video storage requires a large amount of storage capacity. This implies that many disks are required on the same SCSI bus and more SCSI buses need to be used. The Seagate Cheetah performance experiment that is described in Chapter 4, use a total of four SCSI buses and three Cheetahs per Ultra-Wide SCSI bus. When such a configuration is used the total storage capacity is enlarged to almost 110 GB, the throughput per string of disks is 40.9 MB/s and the total bandwidth of the disks is almost 163.7 MB/s.

Unfortunately, current PC hardware is not able to deal with a large number of parallel disks. Currently, the fastest standard SCSI bus runs maximally at approximately 38.1 MB/s, so three Seagate Cheetahs on the same bus saturate such a bus. Also, the PCI bus runs at a theoretical maximum rate of 125.8 MB/s, the PCI throughput is 114.6 MB/s as far as PCI transactions are concerned, but the memory seems to saturate beyond 73 MB/s. Clearly, the aggregate performance of the disks is higher than what can be handled by the computer hardware. So, in order to schedule disks without missing deadlines, hardware needs to be scheduled as well, as has already been observed by Sha *et al.* [92].

To understand how to schedule disks on a SCSI bus, it is important to understand what happens when the bus becomes overloaded. Figure 8.1 shows the cumulative distribution for 1 MB transfers of 1, 2 and 3 disks on the same Ultra-Wide SCSI bus. The figure shows that the performance reduction from 1 to 2 disks is a small one: the reduction is approximately 4 ms, which corresponds to the size of the SCSI-bursts per disk. Since Clockwise currently uses the 90-percentile to determine the service times (*i.e.*, 80 ms per megabyte block), two disks can be scheduled at a rate of 24.7 MB/s for 1 MB transfers.[1]

When, however, 3 disks are used on the same SCSI bus, the transfer times of all requests quickly deteriorate. Instead of a minimum operation time of approximately 70 and 73 ms, the minimum transfer time of a 1 MB block of data from disk now takes approximately 88 ms. The 90-percentile of the requests complete in 96 ms. Hence the maximum aggregate schedulable performance of 3 disks at the same bus is approximately 31.3 MB/s.

[1] Ignoring seek times and an extra rotational delay to deal with unaligned track requests.

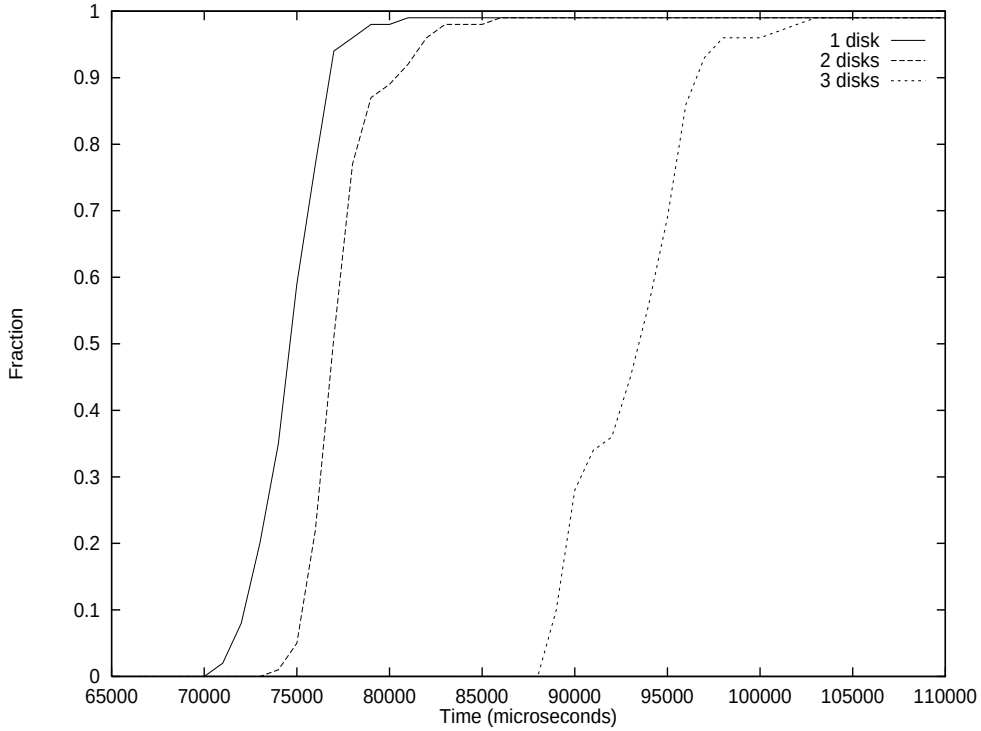

Figure 8.1: 1 MB read transfers for 1, 2 and 3 disks.

The fundamental question to be asked is whether Clockwise needs to schedule on 80 or 96 ms. When Clockwise schedules the disks on 80 ms per megabyte, it acknowledges that the connecting bus can be overloaded and to prevent this from happening only two disks at the same time can be used for data transfers. If, however, Clockwise schedules on 96 ms, it acknowledges that the connecting bus can be overloaded but it anticipates the overload situation by reducing the maximum expected performance from a single disk.

Reducing the expected performance is a questionable solution. Although in the example the situation is not so dramatic, the maximum schedulable performance of a disk is reduced by only 15 %, the situation worsens when more than 3 disks are using the same bus. This means that when more than 2 disks are sharing the same bus, none of the disks can be scheduled at their maximum rate. It also implies that when streams are recorded or played back that have approximately the rate of a single disk, more disks are required to perform such operations.

The alternative of only scheduling two disks at their maximum rate also has a negative side. The third (and possibly fourth, fifth, *etc.*) are not scheduled at all: they are *unschedulable* when real-time streams are admitted for two disks on the same bus. Dynamic partitions that are stored on these disks are temporarily unavailable. This is even the case when the admitted streams require only a small fraction of the available SCSI bandwidth: there is a (small) chance that all three (or four, *etc.*) disks require the SCSI bus at exactly the same time. Further, not even best-effort traffic can be scheduled by the unschedulable disks when no precautions are taken, there is the same risk of simultaneous data transfer from all disks across the SCSI bus.

The only way to schedule the unschedulable disks, is when one or more of the *schedulable* disks indicate that they are idle. Consider that the amount of schedulable disks is d and real-time streams are admitted on all schedulable disks. An altered disk scheduler for an unschedulable disk makes sure that at any instance the amount of *executing* disks e is always less than or equal to d. The altered disk scheduler makes sure that during the execution of a best-effort task on the unschedulable disk, no real-time requests on the schedulable disks become active. This, of-course, can be accomplished by using the ΔL slack time scheduler of the schedulable disks.

When an unschedulable disk needs to execute a best-effort request, it identifies this at an entity called a SCSI-string scheduler. This SCSI-string scheduler oversees all slack time scheduling on all disks that share the same SCSI bus. When one of the schedulable disks enters the idle state or finds slack in the schedule, it notifies the SCSI-string scheduler including its maximum slack time. This SCSI-string scheduler selects a best-effort job to execute on any of the unschedulable disks, and informs the disk driver to execute the request. Upon completion of the best-effort request, the SCSI-string scheduler is notified again, which, on its turn, informs the original schedulable disks of the completion of best-effort task on the schedulable disk. The schedulable disk re-calculates the remaining slack time again. Hence, in some sense the unschedulable disks are *stealing* slack time from schedulable disks.

An alternative to scheduling the disks independently through the ΔL scheduler is to use a combined SCSI-string and disk scheduler that allows *dead time* in the schedule, much as proposed by Audsley *et al.* [5] for preemptive systems. In the proposed *deadline-monotonic* schedulers, the deadlines are not equal to the period of a task. When such a scheduler is used to schedule multiple disks on the same SCSI string, the available SCSI-bus time can be partitioned statically into portions where each disk (or pairs of disks) are assigned SCSI-bus time. Unfortunately the existing deadline-monotonic schedulers that are currently available are preemptive schedulers and work is required to combine the nonpreemptive ΔL scheduler that is presented in this thesis with a deadline-monotonic scheduler.

In Figure 4.17, it is also shown that the memory system becomes a bottleneck when more than approximately 72 MB/s is read from memory. Memory write performance also deteriorates at some point, but since memory writes complete faster than memory reads, the deterioration happens at a later instance. From the figure it seems that there are overload situations when more than 63 % of the available read bandwidth is used.

A detailed study to the memory usage by all devices is required to understand which devices perform concurrent memory operations. When the low memory utilization is caused by concurrent memory accesses, there are two approaches for Clockwise to deal with these problems. The simplest solution is to limit the the maximum memory rate: jobs are only admitted when the resulting memory transfer rate remains below the memory-bandwidth threshold. When higher rates are requested, Clockwise rejects the requests. This solution, however, implies that Clockwise is the only application that makes use of the machine's memory. In reality there are more applications that simultaneously use the memory.

Alternatively, an explicit main memory scheduler can be used to increase the main memory utilization, much like as is proposed in Sha *et al.* [92]. Devices can only access the main memory array when they are granted access to the main memory by means of the main memory scheduler. To schedule devices that make use of the main memory, the SCSI-string scheduler approach

cannot be used anymore because it is impossible to control the memory accesses from all devices by just scheduling disks. Instead, the memory controller itself needs to be changed with support for explicitly scheduled memory accesses. One of the possibilities is to integrate such schedulers in bus switches that are currently developed for the Moby Dick project [36].

In the discussion so-far, networks have not played an important role. However, scheduling networks, such as ATM networks, can in some cases be performed in hardware: network cards that are used today sometimes have support to specify data rates. When such hardware support is absent, a network driver can limit data rates much like as is done in Clockwise for the disks and as is done in some Nemesis network drivers [11]. There is a possibility that the combined load from network and disk is more than what can be handled by the main memory system, in which case disk service times may deteriorate. In such cases it is important to control the network traffic over the local bus.

8.2 Clockwise bandwidth considerations

Chapters 4 and 7 demonstrated the I/O capabilities of single PC based server machine when such a machine is sensibly equipped. PCI buses can be and probably is extended to a 66 MHz, 64 bit bus, a bus that is capable of transferring data at a rate of up to 503.5 MB/s. This is more than enough to saturate both the main memory system and any known network. Unfortunately, the 'length' of this bus in terms of number of available PCI slots is limited, which means that the applicability for large storage systems is limited. Even by using Fibre-Channel like SCSI networks, which have an aggregate network bandwidth of 100 MB/s, only a part of the PCI bus bandwidth can be employed for I/O bandwidth since it is hardly possible to use more than a single network card because of the limited number of PCI slots.

The only way to improve the I/O bandwidth of a Clockwise server is by parallelizing the service over multiple machines or *nodes*. Each of the independent Clockwise nodes in such a network is capable of delivering up to the network speed of data. When a collection of nodes work as a single service connected by many (parallel) routes through a network, serious sustained bandwidth becomes available. Such a structure is an alternative to VxFS' approach that tries to achieve high bandwidths through a single server [98]. The advantage of using multiple machines, is that the service may become more fault tolerant and from an economical standpoint, cheaper hardware can be used.

Admittedly, currently there are not many applications that require such a high throughput. An example application is an HDTV studio. Such a studio may require high real-time throughput: a single uncompressed HDTV stream consumes approximately 112 MB/s and when a number of streams are combined into a new single stream, a multiple of this bandwidth is required. Future applications such as virtual-reality engines, large corporate information systems or scientific super-computer applications may require such high data rates.

The reason for discussing high-speed storage in this thesis is that Clockwise components can easily be used in a distributed storage service to deliver more than 1 GB/s throughput. This is done by extending the dynamic partition model across multiple Clockwise nodes and allowing the distributed dynamic partition to be accessed as if it is stored on a single disk.

Currently, a dynamic partition is described by a Clockwise *dpnode* as a list of disk identifications and block addresses on disk. It is, of course, conceivable that these disk identifications are mapped instead of to a local machine's disk controller and disk unit to any machine's disk controller and disk unit that is connected by a network. A request for a non-local dynamic partition block, implies sending a request across the network instead of to a disk controller.

For a distributed version of Clockwise the meta-data management is not much different from a centralized Clockwise. However, in light of failures during an update of, for example, the superblock, the protocol required to do so needs to make sure that all or none of the updates are executed. Two-phase, or three-phase commit protocols are possible candidates for such operations. Fortunately, superblocks do not have to be updated frequently: only when new disks are added or old disks are removed from the (distributed) Clockwise server, or when the address of the DPTAB changes.

Clockwise DPNODEs can be updated regularly. For every new block that is allocated, or deleted from the dynamic partition, a new version of the DPNODE is written. The DPNODE currently lives on a single disk. To use DPNODEs in a distributed Clockwise, a single Clockwise server, an DPNODE *master*, acts as intermediary between the (remote) client application and the DPNODE. To add or remove a block to or from a dynamic partition, a client request this operation to be completed by the DPNODE master. The DPNODE master forwards the operation to the requested Clockwise server and disk. When successful, the block is added to or removed from the dynamic partition.

Currently Clockwise reconstructs block allocation bitmaps by reading in all of the DPNODEs and determining which blocks are in-use and which are free. This algorithm works fine for a centralized Clockwise, but when Clockwise is used in a distributed manner, some of the servers may be (temporarily or permanently) unavailable. These unavailable Clockwise servers can be the DPNODE master for blocks that are located on other Clockwise servers, so when only the local DPNODEs are scanned not all of the allocated blocks are found. To deal with this problem, the allocated block bitmaps are stored separately on disk as well. This means that even when a remote Clockwise server is unavailable, a local Clockwise server is able to initialize itself without endangering other dynamic-partition's blocks.

Distributed Clockwise allows client applications to read and write data directly from one of the Clockwise servers without contacting the DPNODE master. When a dynamic partition is opened, the DPNODE master replicates a copy of the DPNODE to all Clockwise servers that hold the data for the dynamic partition, along with a secret that is only known by the Clockwise server and the client. The client presents this secret on each request and the Clockwise server uses this secret to authenticate the client. Such a structure allows a Clockwise server to be used as a *Network Attached Secure Disk* (NASD) [27].

8.3 Clockwise for Linux

The Nemesis operating system is a prototype operating system that is developed as part of the ESPRIT Pegasus project by the University of Cambridge. Since Clockwise is also developed as part of the Pegasus project, the first implementation of Clockwise runs on the Nemesis operating

system.

Nemesis is fundamentally different from existing operating systems (see Section 5.1). Porting a Nemesis application to another (existing) operating system requires that the entire application needs to be re-written for the other operating system. This is not different for Clockwise.

To run Clockwise on, for example, the UNIX operating system [82], a many modifications are required. There are two basic problems when implementing Clockwise on a UNIX machine: the CPU needs a true real-time scheduler and the disks need to be scheduled explicitly. The current UNIX CPU scheduler is usually only a prioritized round-robin CPU scheduler that has no notion of real-time processes. If there is support for real-time scheduling at all (such as in real-time Linux [7]), then most likely the real-time system is implemented *underneath* the UNIX functionality instead of inside UNIX. Scheduling UNIX disks is usually only done to optimize small disk accesses. An overview is given of how to port Clockwise functionality to Linux.

8.3.1 Linux CPU scheduling

The simplest modifications are the changes to CPU scheduling. Whenever the Linux CPU scheduler needs to pick a task to run, it executes the task with the highest priority on the *runnable* queue. Whenever a process becomes runnable, the *nice* value determines the position of the process in the queue. To support, for example, a Nemesis type of scheduling [83], the insertion routine needs to use deadlines rather than the nice values to determine the position of a runnable process in the run queue.

To support EDF scheduling, Linux needs to be extended with an EDF admission test. Whenever a process requests real-time service, it presents a QoS setting with the requested period and slice. Linux uses this information to execute the standard EDF schedulability test and when the task is accepted it runs as a real-time process.

The Linux process structure needs to be extended with support for maintaining process *periods*, allocated and available *slices* and process *deadlines*. When a process is made runnable it is given a slice that is initialized to its requested CPU slice. The actual CPU usage is subtracted from the allocated slice on every timer interrupt. When the available slice becomes zero, the process is delayed until its next release time. If a process is given too much CPU time it can yield the CPU and the process is delayed until its next period.

Linux is not Nemesis: Nemesis accounts *most* if not all resource usage to the user domains that caused the resource usage, whereas this is virtually impossible on Linux. By integrating some of the Nemesis scheduling techniques inside Linux, it is expected that Linux will behave more predictable with regard to handling continuous-media data streams. However, in all cases, the Nemesis approach for dealing with continuous-media data is superior to the Linux approach.

8.3.2 Linux disk scheduling

Currently existing device drivers do not implement any of the functionality that is required to schedule disk requests explicitly. Instead, current UNIX disk drivers only implement a FCFS scheduling queue, possibly augmented with an optimization policy to implement elevator seeks [50, 90].

To introduce dynamic partitions and a ΔL scheduling principle to the Linux kernel, a new disk-device driver is required. This driver introduces all of the Clockwise principles to a Linux machine. The Linux/Clockwise driver is given a collection of disk drivers to operate on and it uses the drivers to read and write disk blocks from drives. The Clockwise driver itself does not implement any of the reading and writing itself.

The advantage the Linux/Clockwise driver approach is that existing drivers do not have to be changed in order to support explicitly scheduled disks. This means that such a structure allows the scheduling of any type of disk controller provided service times can be predicted. In particular, Nemesis/Clockwise only support the scheduling of NCR based SCSI devices; Linux/Clockwise can also schedule other I/O devices such as IDE drives.

When Linux/Clockwise starts, a number of daemons (one per disk) are activated that implement the EDF queue for a disk. The Linux/Clockwise driver queues user requests in the EDF queues, and the daemons extract the requests and activate the original disk drivers. These daemons run under the above described real-time CPU scheduling regime. The Linux/Clockwise part is responsible for calculating the release times and deadlines, the daemons are responsible for the actual execution of the requests on the disks. To make sure that only scheduled traffic arrives in the original drivers, the only way to access a ΔL scheduled disk is through the Clockwise driver.

Linux/Clockwise dynamic partitions are not different from the dynamic partitions that are used by Nemesis/Clockwise. A dynamic partition can span multiple disks and dynamic partitions can grow and shrink in size. In fact, the data structures that are used by both versions of Clockwise are identical and the Clockwise disks can be shared between the Linux and Nemesis version of Clockwise.

UNIX user applications can access Clockwise dynamic partitions through two methods. Each dynamic partition is represented by a UNIX minor device and each dynamic partition can be accessed either through the character or block device interface. When the block device is used, the dynamic partitions are used to store UNIX (best-effort) file systems. When the character interface is used, the dynamic partitions are used for, for example, continuous-media data storage. Each of the interfaces can be explicitly scheduled and QoS service parameters can be associated with each opened dynamic partition.

A prototype Linux/Clockwise version now exists as a Linux module and is used for experimentation.

8.4 Service-times predictions

The process of determining how fast a disk is, as is shown in Chapter 4, usually a tedious task. Clockwise currently is able to schedule two disk types explicitly: the Quantum Atlas-II and Seagate Cheetah disk. Measuring these disks and understanding what they are doing, is a time-consuming task: a large number of I/O requests need to be executed on a disk and the results need to be analyzed afterwards.

To allow Clockwise to deal with a number of different types of disks, a semi-automated method is required to determine the service-times during the execution of Clockwise. If, for every disk type that is available, a person needs to measure the actual performance of disks,

many man-months are spent retrieving the parameters from the disk. Ideally, when a new disk is inserted into a Clockwise system, Clockwise figures out the disk behavior by itself. By issuing a (large) number of I/Os to the new disk, it can construct a disk-scheduling table, which it later uses to predict service times.

To predict service times Clockwise only needs to know the block service time and the seek performance. The block service time depends on the rotational speed and the number of sectors that are formatted on a track. Since the rotational speed is usually known, Clockwise only needs to find out how many sectors that are per track and whether a disk uses more than a single disk zone. To retrieve zoning information from a disk, Clockwise can use a similar technique as is described in Section 4.3 by using the SCSI translate address operation. By establishing the zone information, the physical layout of the disk is known.

Once the zone information is known, Clockwise can retrieve seek timing information by executing a number of seek operations on the new disk. As is shown in Section 4.3, the only operation Clockwise needs to perform is to instruct the disk heads to move between two disk addresses. The path between the minimum and maximum seek time consists usually of two distinct phases: a startup phase and a linear phase (*cf.* Figure 4.5).

To determine the seek behavior for the new disk, Clockwise first determines the minimum and maximum seek time by seeking only a single cylinder and a full stroke. Next, Clockwise determines the seek times of transfers that are half the length, and depending on the desired precision in seek times, it continues the process for a quarter (or even smaller distances) of the maximum seek distance.

To learn of the service times for disk requests in disk zones, Clockwise enters a mode where it starts measuring the disk. It first assumes that the disk behaves much like what can be expected from the disk's physical parameters. Also, Clockwise keeps statistics and a log of the actual service times during execution of user requests. Since the seek behavior is already known, the actual disk transfer times are measured. The user requests that Clockwise executes during this mode can either be generated by ordinary Clockwise clients, or by a specialized utility that allows Clockwise to learn of the performance of the disk.

To trim the service-time parameters, Clockwise makes available the performance characteristics to an administrator. This person analyzes the performance characteristics and installs the parameters for later use when no anomalies are found. If, however, performance anomalies are found, the administrator needs to execute performance experiments to learn the reason of the performance anomalies.

8.5 Tertiary storage

Clockwise currently has no support for tertiary-storage devices. Such integration is important because secondary storage is usually an order of magnitude more expensive than tertiary storage. So, from an economical standpoint it is wise to store continuous-media data from Clockwise that is not accessed frequently on tertiary storage. Continuous-media data is retrieved from tertiary storage and cached on secondary storage on demand. Moving continuous-media data from secondary storage to tertiary storage is called *demotion* of data and from tertiary storage to

secondary storage is a called *promotion* of data [104].

Tertiary storage is usually organized around a robotic device (*e.g.* a CD juke-box). The robotic device consists of a number of medium readers and writers and a vast amount of removable storage, such as CDs, tapes or DVDs. Compared to the availability of secondary storage devices, the medium readers and writers are a scarce resource – there are only a few of them.

Data that is stored on tertiary storage can be directly or indirectly transmitted to clients. When data is sent directly to a client, data is not promoted to secondary storage before it is sent to a client. When data is sent indirectly, the data is first promoted to secondary storage and the promoted data is transmitted to the client.

There are several reasons why it is better to send data indirectly. First, if tertiary storage is slower than the requested transfer rate, it needs to be promoted anyway to cope with the higher data rates or else smooth delivery of data is not possible. If the requested rate is lower than the bandwidth of tertiary storage, it is better to promote the data before delivering it to the client to increase the availability of medium readers and writers.

Only when the requested rate equals the maximum rate of the medium reader or writer, there is no point in promoting the data to secondary storage to increase the availability. If, however, the same data is requested a short while later for a second time, promoting the data to secondary can improve the availability of tertiary storage: the data can still be cached on secondary storage.

The Huygens laboratory at the University of Twente uses a DAX CD juke-box for tertiary storage. This juke-box holds 720 CDs and contains three 12 speed CD readers and one 8 speed CD writer. A robot arm inside the juke-box can be instructed to move CDs from storage trays into CD readers or writers,[2] which can be read through a SCSI bus. The SCSI bus connects all CD readers and writers. The robot arm is given commands through a serial connection between the Clockwise machine and DAX.

8.5.1 Demotion and promotion of data in Clockwise

Clockwise allows clients to demote data on demand and promote data automatically between the DAX CD juke-box and the Clockwise server. A client can select a Clockwise dynamic partition and ask Clockwise to move data to an empty CD. To promote data, Clockwise controls the DAX to load a CD into an empty reader and to upload the data from CD into a (cache) dynamic partition.

In principle, all free Clockwise disk space is available to cache tertiary data. Clockwise maintains a LRU table with the cached dynamic partitions and whenever free memory is required either for ordinary dynamic partition storage or to cache new data from the DAX, the oldest cached dynamic partitions are discarded.

When there is no or hardly any free cache space available on secondary storage, Clockwise reverts to playbacks directly from the CD readers themselves. This policy reduces the availability of the tertiary storage, since the scarce CD readers are used less efficiently.

When Clockwise can demote data automatically from secondary storage to tertiary storage, Clockwise can always maintain enough free space to cache data from CD. This, however, implies that Clockwise needs to know of the contents and access pattern of the dynamic partitions. Since

[2] At least, when the DAX has not decided to Frisbee with the CDs.

Clockwise currently does not have knowledge of the contents of a dynamic partition, it does not know if there will be many updates to the dynamic partition in the future. Updating a CD usually implies that a new CD needs to be burned.

8.5.2 Archival support

Tertiary-archival storage is used to archive best-effort UNIX file systems. Daily backups are made to preserve the state of a file system on a particular day. The backup utility examines the contents of an UNIX file system and those files that have been modified are backed up on tertiary storage (much like what the UNIX *dump(8)* utility does).

The Clockwise *dump* utility is a simple utility that traverses a VFS file system and identifies all files that are modified since the last backup. It copies all the new files including the directories that lead to the file to tertiary storage. Those directories and files that have not changed since the last backup contain a pointer to the last backup of the file or directory. The backup itself is then made available again for clients. The backup utility itself does not yet exist for Clockwise and the backup mechanism itself is not new: Plan 9 from Bell-Labs uses a similar backup approach [77].

8.6 Summary

Current disk technology can, when it is used in parallel, easily outperform current (PC) architectures. A single Quantum Atlas-II disk, for example, outperforms a Fast SCSI-2 bus and three Seagate Cheetahs outperform Ultra-Wide SCSI bus. Also, when more than six Seagate Cheetahs are used that are connected by three SCSI buses, the machine itself becomes a bottleneck: current memory seems to saturate when more than 72 MB/s is transferred through a machine.

There are three approaches in dealing with the overload situations. When a number of disks are used in a parallel fashion, the worst-case request timings are used to determine the task's scheduling parameters. Although this approach has the advantage that all disks can be used at the same time without any restriction, this approach has two drawbacks: the maximum utilization per disk can be low and this approach does not solve the real problem of scheduling data transfers through a machine.

A second approach is not to allow overload situations to occur. In this case, the disks are only scheduled when it is known on beforehand that scheduling the disks does not lead to overload situations. In the example of three Seagate Cheetahs that share a Ultra-Wide SCSI bus, only two disks are scheduled at a time. The ΔL scheduler can be extended with a protocol to share the pre-calculated slack time over all the disks that share the same SCSI bus.

However, the fundamental problem is that inside a machine, hardware disk schedulers determine the order in which all devices share the machine's infrastructure. So, the only way to orchestrate data streams well is by changing these policies to do 'something' sensible instead of current round-robin approaches. This solution is a non-trivial one and requires a complete redesign of current hardware.

Although a single Clockwise server can deliver a substantial throughput in a real-time manner, some applications require more throughput. By extending some of the key data structures

that are used by Clockwise to include network addresses, an extension is presented that allows Clockwise to be used in a distributed manner. The collection of Clockwise servers can sustain higher data rates than each of the single servers can deliver on its own.

Currently Clockwise runs as a Nemesis application. The problem with Nemesis is that it is only a prototype operating system and has stability problems. A file server, however, is precisely the type of service that must *always* work. To improve on the stability of the system as a whole, it is therefore important to migrate Clockwise to a UNIX platform. An approach is presented to perform this migration. The migration includes changing the UNIX CPU and disk schedulers.

Clockwise is currently able to schedule two disk types: the Quantum Atlas-II disks and Seagate Cheetah disks. Measuring the behavior of these disks proved to be a tedious task. For each new disk that needs to be scheduled by Clockwise, these measurements need to be repeated. An approach is presented that allows Clockwise to derive most of the scheduling parameters automatically.

Lastly, an overview is given how to integrate tertiary storage devices in Clockwise.

Chapter 9

Summary and conclusions

This thesis addresses the problem of the storage of real-time (continuous-media) file data and unscheduled best-effort file data through one service. To do this correctly, one needs an understanding of the nature of best-effort and continuous-media data, one needs to know how to use storage hardware efficiently and one must know how to schedule real-time and unscheduled requests on disks.

This thesis starts out by describing the nature of continuous-media data traffic and best-effort (file-system) traffic. Continuous-media data is bulky and to playback or record continuous-media data streams, isochronous service is required. Capturing and rendering of multiple continuous-media data streams only succeeds if the underlying hardware is used efficiently. To support isochronous data transfers, the storage service needs to support (some form of) real-time request scheduling. On the other hand, best-effort UNIX file-system traffic is optimized by minimizing the request latencies. These goals seem orthogonal.

This thesis demonstrates that commodity hardware can be used efficiently and sustain high data rates. Measurements show that a standard Pentium-Pro based PC can sustain data rates up to 100 MB/s. However, the same measurements indicate that when more than approximately 70–80 MB/s is sent through a PC, devices start to interfere with each other. Current state-of-the-art networks can transfer data at a rate of 155 Mb/s, which implies that a single PC-based storage server can easily saturate such a network. Faster network technology is required to appreciate the I/O capabilities of a well-configured PC.

The key to high throughput on a PC is that all data transfers are performed by the DMA mechanism of the I/O cards and that the disk executes large sequential transfers. Based on these efficiency principles, a storage system called Clockwise is designed: it uses the DMA capabilities of I/O cards to move data around between network and disk. Also, Clockwise organizes the storage space in *dynamic partitions* to group large disk blocks. Currently dynamic partition blocks are 1 MB large. Dynamic partitions can span multiple disks and can grow and shrink in size.

Clockwise is currently used for two storage applications: continuous-media applications and best-effort UNIX file-system applications. Continuous-media applications store continuous-media data in the dynamic partitions and best-effort UNIX file systems organize UNIX file systems inside the dynamic partitions.

Continuous-media applications are applications that orchestrate the transfer of continuous-media data between network and disk on a Clockwise server. When audio or video data needs to be played back, a continuous-media application instructs Clockwise to load audio or video from disk into network packets. When the packets are read into memory, the continuous-media application instructs the network controller to transmit the packets to the client. Alternatively, when a continuous-media data is recorded, the continuous-media application gathers a dynamic partition block worth of audio or video in memory and instructs Clockwise to store the received network packets consecutively in a dynamic partition block.

Clockwise memory is used to cache data for best-effort UNIX file systems and to buffer continuous-media data for continuous-media applications. Clockwise balances memory requests between the various clients and divides the memory in a fixed (guaranteed) part and variable part. Clockwise never revokes fixed memory, but can revoke variable memory buffers.

The devices that play a critical role in a mixed-media file system are the disks. To guarantee that all deadlines are met by the (continuous-media) real-time applications, Clockwise uses a *nonpreemptive Earliest Deadline First* (EDF) scheduler. This scheduler uses an admission test that guarantees that all deadlines are met, even when possibly long running disk requests cannot be preempted. The key to the admission test is that it iterates all possible release times and makes sure that any combination of requests is schedulable.

The nonpreemptive EDF schedulability analysis is extended with *slack time* calculations. This slack time, which is called ΔL, is proven to be the minimum slack time between the completion time of *any* real-time task invocation and its deadline. Given that this time is always available after each invocation, all invocations can also start ΔL later than is originally planned. Clockwise uses this ΔL to give precedence to unscheduled best-effort requests. So, even when there are real-time requests queued, unscheduled best-effort requests, *i.e.*, requests without a deadline, can be executed *before* queued real-time requests.

A set of measurements are presented where ΔL scheduling is compared to other existing scheduling methods. Through the scheduling experiments it is shown that only ΔL scheduling can guarantee deadlines. The other schedulers, USD's EDF and Symphony's *Latest Start Time* (LST) scheduler, can miss arbitrary deadlines since these schedulers do not consider the minimum-slack time and the nonpreemptiveness of disks. Also, when the best-effort latencies are compared for all schedulers, it is demonstrated that for light to moderate real-time loads, the best-effort latencies are comparable for the LST and ΔL schedulers, with a slight advantage for the LST scheduler. When, however, the real-time load is high, the latencies of best-effort requests that are scheduled by a LST scheduler quickly increase because the scheduler does not use spatial locality: precious slack time is wasted on disk seek operations between best-effort and real-time dynamic partitions. The ΔL scheduling does not suffer from this effect because it executes all of the real-time requests in a single burst.

The scheduling experiments are validated with online experiments in a real Clockwise. As is shown by the simulations, three parallel Quantum Atlas-II disks with an aggregate bandwidth of 25.3 MB/s can sustain an admitted real-time load of up to 21 MB/s *and* execute a peak best-effort load of 4,000 I/O requests per minute. Admittedly, with such rates the best-effort latencies are not low and large best-effort queues build up. However, none of the real-time deadlines are missed when the ΔL scheduler is used to schedule the disks.

A pre-condition for ΔL scheduling is that disk service times can be predicted. The ΔL scheduler uses the predicted service time for admission tests and to predict how long a best-effort request takes. Currently, Clockwise calculates service times based on raw performance measurements. A set of performance experiments are performed on top of Clockwise to learn if the actual service times of disk requests is comparable to what is measured in the raw performance experiments. Because dynamic partition blocks are not aligned to track boundaries, the predicted service times are underestimated. The raw service times are extended with the time to perform a single rotation to cope with the unaligned dynamic partition blocks. An optimization technique with a Clockwise disk cache allows efficient data transfers for sequentially read dynamic partitions.

When Clockwise uses multiple disks in parallel and it does not overload the SCSI or PCI bus, there is an almost linear speedup in aggregate performance. It is shown that six Seagate Cheetah disks on three Ultra-Wide SCSI buses can transfer data up to 80 MB/s. When the SCSI or PCI bus is overloaded, the service times become unpredictable. The reason for the unpredictability is that there are many hidden (round-robin) policies inside PC hardware.

There are two ways to solve the hidden policy problem. One can experimentally determine the upper transfer rate and do not allow admissions above this rate, which is Clockwise's current approach. Alternatively, the hidden policies can be changed to something sensible. The latter option is favorite and may be the only way to construct a general purpose machine that supports real-time scheduling. Experiments with an explicitly scheduled SCSI bus are already underway.

To enlarge the sustained throughput of Clockwise, a design is presented on the construction of a distributed Clockwise. Such a construction is not a difficult task: it requires the extension of disk identifications with Clockwise server addresses and it requires the introduction of one extra data structure on disk.

Clockwise exist and runs on a real-time operating-system kernel called Nemesis. Nemesis has the disadvantage that it is still a prototype system, and is not yet stable enough to host a departmental file service. To improve Clockwise's stability, Clockwise is currently being ported to standard UNIX platforms.

All in all, it is surprising to notice that the scheduling approach that is presented in this thesis was not used before. The nonpreemptive scheduling technique is published almost a decade ago, but to date and to my knowledge it is not used to schedule disks, if at all; even though the scheduling technique seemed to be designed to schedule disks. To date, disks were either scheduled by some sort of *Time-Division Multiplexer* (TDM) scheduler or by schedulers that do not account for the nonpreemptiveness of disks. By trying to understand the original nonpreemptive scheduling paper, it became apparent *why* ΔL scheduling is superior: it also gives a handle on the available slack time in a schedule and hence the schedulability of unadmitted best-effort requests.

Peter Bosch, 1999.

Bibliography

[1] Roalt Aalmoes and Peter Bosch. Overview of Still-picture and Video Compression Standards. Pegasus 95-03. University of Twente, December 1995.

[2] David P. Anderson, Yoshitomo Osawa, and Ramesh Govindan. The Continuous Media File System (Real-Time Disk Storage and Retrieval of Digital Audio/Video Data). *USENIX Conference Proceedings* (San Antonio, TX), pages 157–64. USENIX, Summer 1992.

[3] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, **14**(1):41–79, February 1996.

[4] ANSI. *Draft proposed American National Standard for information systems – Small Computer System Interface-2 (SCSI-2)*, 1991.

[5] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software* (Atlanta, GA). IEEE, 1991.

[6] Mary G. Baker, John H. Hartman an Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Pacific Grove CA (USA)), number 25(5) in Operating Systems Review, pages 198–212, October 1991.

[7] M. Barabanov and V. Yodaiken. Introducing Real-Time Linux. *Linux Journal*, **34**:19–23, 1997.

[8] P. R. Barham, M. D. Hayter, D. R. McAuley, and I. A. Pratt. Devices on the Desk Area Network. *submitted to Journal on Selected Areas in Communications*, 1994.

[9] Paul Barham. A Fresh Approach to Filesystem Quality of Service. *7th International Workshop on Network and Operating System Support for Digital Audio and Video* (St. Louis, Missouri, USA), pages 119–128, May 1997.

[10] Paul Ronald Barham. *Devices in a Multi-Service Operating System*. PhD thesis. Churchill College, University of Cambridge, July 1996.

[11] Richard Black. *Explicit Network Scheduling*. PhD thesis, published as Technical report TR361. University of Cambidge, April 1995.

[12] Willian J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed Schedule Management in the Tiger Video Fileserver. *Proceedings of 16th ACM Symposium on Operating Systems Principles* (Saint-Malo, France). Published as *Operating Systems Review*, **31**(5), October 1997.

[13] Peter Bosch. A cache odyssey. M.Sc. thesis, published as Technical Report SPA–94–10. Faculty of Computer Science/SPA, Universiteit Twente, the Netherlands, 23 June 1994.

[14] Peter Bosch. Pegasus deliverable 2.2.2: Profiling tools for Nemesis. Pegasus Deliverable 2.2.2. University of Twente, Netherlands, October 1998.

[15] Peter Bosch and Sape J. Mullender. Cut-and-paste file-systems: integrating simulators and file-systems. *USENIX* (San Diego, CA), January 1996.

[16] Giorgio C. Buttazzo. Chapter 5, Fixed-Priority Servers. In *Hard Real-Time Computing Systems*, pages 109–48. Kluwer, 1997.
[17] Giorgio C. Buttazzo. Chapter 6, Dynamic Priority Servers. In *Hard Real-Time Computing Systems*, pages 149–81. Kluwer, 1997.
[18] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy K. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, **26**(2):145–185, June 1994.
[19] ATM Forum Technical Committee. *ATM User-Network Inteface Specification*, September 1994.
[20] Digital Equipment Coorporation. *TURBOchannel Hardware Specification*, Januari 1993.
[21] Digital Equipment Corporation. ATM Connectivety: ATMworks 350 Adapter. Digital Equipment Corporation, 1997. www.networks.digital.com:80/html/products_guide/adapt13.html.
[22] Digital Equipment Corporation. *DECchip 21050 PCI-to-PCI Bridge Data Sheet*, December 1994.
[23] Intel Corporation. *Pentium Pro Family Developer's Manual – Volume 1: Specifications*, 1996.
[24] Intel Corporation. *Intel 440FX PCI set 82441FX PCI and memory controller (PMC) and 82442FX Data Bus Accelerator (BDX)*, May 1996.
[25] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. Logical disk: a simple new approach to improving file system performance. Technical report IR-325. Dept. of Mathematics and Computer Science,Vrije Universiteit, Amsterdam, 1993. Also MIT/LCS/TR-566 at MIT.
[26] Robert M. English and Alexander A. Stepanov. Loge: A Self-Organizing Disk Controller. *USENIX Conference Proceedings* (San Francisco, CA), pages 237–52. USENIX, Winter 1992.
[27] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File Server Scaling with Network-Attached Secure Disks. SIGMETRIC *'97* (Seattle, WA, USA), pages 272–284. ACM, 1997.
[28] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Conf. Proc. of the USENIX 1995 Tech. Conf. on UNIX and Advanced Compt. Sys.* (New Orleans), pages 201–12. Usenix Assoc., 16–20 Jan. 1995.
[29] PCI Special Interest Group. *PCI Local Bus Specification*, April 1993.
[30] C.-C. Han and Kwei-Jay Lin. Scheduling Distance-Constrained Real-Time Tasks. IEEE TC Real-Time Systems, pages 300–308. IEEE, December 1992.
[31] Steven Hand. USD *stability problems – personal communication*, 1998.
[32] Michael Gonzalez Harbour, Mark H. Klein, and John P. Lehoczky. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. *Real-Time Systems Symposium*, IEEE TC Real-Time Systems, pages 116–28, 1991.
[33] Lynda Hardman. *Modelling and Authoring Hypermedia Documents*. PhD thesis. Universiteit van Amsterdam, Netherlands, March 1998.
[34] John H. Hartman. Sprite file system configuration, March 1995. personal communication.
[35] John H. Hartman. *Using the Sprite file system traces*, Version 1.0. University of California, Berkeley, CA 94720, USA, may 1993.
[36] Paul J.M. Havinga and Gerard J.M. Smit. The design of a system archictecture for mobile multimedia computers. Technical report. University of Twente, Netherlands, (to appear in) 1999.
[37] Eoin Hyden. *Operating System Support for Quality of Service*. PhD thesis. University of Cambridge, February 1994.
[38] IEEE. *Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language]*, 7 December 1990.
[39] CrosStor Software Inc. CrosStor Volume Manager. http://www.crosstor.com. CrosStor Software Inc., 1999.

[40] Micron Technology Inc. Designing for high performance with synchronous dram modules. TN-48-03. Micron Technology Inc., 1999.
[41] Micron Technology Inc. Choosing the right SYNCBURST SRAM. TN-58-06. Micron Technology Inc., August 1995.
[42] Micron Technology Inc. Evolutionizing the DRAM business. TN-04-39. Micron Technology Inc., December 1995.
[43] Quantum Inc. *Atlas II XP32275/XP34550/XP39100 SCSI Hard Disk Drives Product Manual*, unknown.
[44] Seagate Technology Inc. *Cheetah 9 Family: ST19101N/W/WD/DC Product Manual, Volume 1*, April 1997.
[45] Sun Microsystems Inc. Network File System specification. RFC1094. DDN Network Information Center, SRI International, Menlo Park, CA.
[46] Symbios Logic Inc. *SYM53C810A PCI-Ultra SCSI I/O Processor*, July 1996.
[47] Symbios Logic Inc. *53CF94/96-1 Fast SCSI Controller*, June 1993.
[48] Symbios Logic Inc. *Symbios Logic PCI-SCSI I/O Processors*, June 1997.
[49] Symbios Logic Inc. *SYM53C875 PCI-Ultra SCSI I/O Processor*, September 1996.
[50] David M. Jacobson and John Wilkes. Disk scheduling algorithms based on rotational position. Technical report HPL–CSP–91–7rev1. Hewlett-Packard Company, 16 February 1991.
[51] Pierre G. Jansen and Rene Laan. The Stack Resource Protocol based on Real-Time Transactions. *IEE Proceedings Software*, 1999.
[52] Paul Wenceslas Jardetzky. *Network file server design for continuous media*. PhD thesis. Computing Laboratory at the University of Cambridge, Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, England, April 1992. Techical report no. 268.
[53] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On Non-Preemptive Scheduling on Periodic and Sporadic Tasks. *Real-Time Systems Symposium*, IEEE TC Real-Time Systems, pages 129–139, 1991.
[54] Daniel I. Katcher, Hiroshi Arakawa, and Jay K. Strosmider. Engineering and Analysis of Fixed Priority Schedulers. Technical report. Carnegie Mellon University, 1993.
[55] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. *1986 Summer USENIX Technical Conference* (Atlanta, GA, June 1986), pages 238–47. USENIX, June 1986.
[56] Jan Korst. Random Duplicated Assignment: An Alternative to Striping in Video Servers. ACM *Multimedia '97* (Seattle, Washington), pages 219–26, 1997.
[57] Jan Korst, Emile Aarts, and Jan Karel Lenstra. Scheduling Periodic Tasks with Slack. INFORMS *journal of computing*, **9**(4):351–62, Fall 1997.
[58] Butler W. Lampson. Hints for Computer System Design. *Proceedings of 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, New Hampshire). Published as *Operating Systems Review*, **17**(5):33–48, October 1983.
[59] Didier Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, **34**(4):46–58, April 1991.
[60] John P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *Real-Time Systems Symposium*, IEEE TC Real-Time Systems, pages 201–9, 1990.
[61] John P. Lehoczky and Sandra Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. *Real-Time Systems Symposium*, IEEE TC Real-Time Systems, pages 110–23, 1992.
[62] John P. Lehoczky, Liu Sha, and Ye Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour. *Real-Time Systems Symposium*, IEEE TC Real-Time Systems, pages 166–71, December, 1989.

[63] Vassili Leonov. *Linux Motion Video Project*, 1998.

[64] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication*, **14**(7):1280–97, 1996.

[65] Ian Leslie, Derek McAuley, and Sape J. Mullender. Pegasus - operating system supports for distributed multimedia systems. *SIGOPS*, **27**(1):69–78, January 1993.

[66] C.L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, **20**(1):46–61, January 1973.

[67] Paul Keith Lougher. *The Design of a Storage Server for Continuous Media*. PhD thesis. Lancaster University, Sep. 1993.

[68] Joseph Y.-T. Lueng and Jennifer Whitebread. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, **2**:237-250, 1982.

[69] Yoshifumi Manabe and Shigemi Aoyagi. A Feasibility Decision Algorithm for Rate Monotonic and Deadline Monotonic Scheduling. *Real-Time Systems*, **14**:171-181. Kluwer Academic Publishers, 1998.

[70] Christopher Marahan. Clockwise, 1985. Motion picture.

[71] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[72] John McTiernan. The Hunt for Red October. Paramount, 1990. Motion picture.

[73] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. *1996 Winter Usenix conference* (San Diego, January 1996), pages 279–94. Usenix Association, January 1996.

[74] John Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. *Proceedings of the 10th Symposium on Operating System Principles*, pages 15–24, December 1985.

[75] Nick Park. Wallace and Grommit: A Close Shave. BBC, 1995. Aardman Animations.

[76] D.A. Patterson, G. Gibson, and R.H. Katz. A case for redundant arrays of inexpensive disks (RAID). *International conference of Management of Data (SIGMOD)*, pages 109–116. ACM, 1988.

[77] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. *Proceedings of Summer UKUUG Conference*, pages 1–9, July 1990.

[78] I. Pratt. *ATM camera V1*, ATM Document Collection 2 (The Orange Book). University of Cambridge, February 1993.

[79] Ian Pratt and Paul Barham. *The ATM Camera V2 (AVA200)*. University of Cambridge, 15 March 1994.

[80] P. Venkat Rangan and Harrick M. Vin. Designing file systems for digital audio and video. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 81–94. Association for Computing Machinery SIGOPS, 13 October 1991.

[81] A.L. Narasihma Reddy and Jim Wyllie. Disk scheduling in a multimedia I/O system. *ACM Multimedia 93*. ACM, 1993.

[82] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, **17**(7):365–75, July 1974.

[83] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, published as TR-376. University of Cambridge, April 1995.

[84] Timothy Roscoe. Linkage in the Nemesis Single Address Space Operating System. Technical report. Univ. of Cambridge, May 1994.

[85] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Pacific Grove CA (USA)), number 25(5) in Operating Systems Review, pages 1–15, October 1991.

[86] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, pages 17–28, March 1994.

[87] Chris Ruemmler and John Wilkes. UNIX Disk Access Patterns. *1993 Winter Usenix conference* (San Diego, CA), pages 405–20. Usenix Association, January 1993.

[88] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. *USENIX Association Summer Conference Proceedings of 1985* (11-14 June 1985, Portland, OR), pages 119–30. USENIX Association, El Cerrito, CA, 1985.

[89] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. *USENIX Technical Conference Proceedings* (San Diego, CA), pages 307–26. USENIX, Winter 1993.

[90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling Revisited. *USENIX Conference Proceedings* (Washington, D.C.), pages 313–24. USENIX, 22-26 January 1990.

[91] Margo Ilene Seltzer. *File system performance and transaction support*. PhD thesis. University of California, 1992.

[92] Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. *Real-Time Systems Symposium*, IEEE TC Real-Time Systems, pages 181–91, 1986.

[93] Prashant J. Shenoy, Pawan Goyal, Sriram S. Rao, and Harrick M. Vin. Symphony: An Integrated Multimedia File System. http://www.cs.utexas.edu/users/dmcl. University of Texas at Austin, 1996.

[94] Prashant J. Shenoy and Harrick M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. http://www.cs.utexas.edu/users/dmcl. University of Texas at Austin, 1996.

[95] Ralf Steinmetz. Architecture and Protocols for high-speed networks. In Effelsberg Otto Spaniol, Danthine, editor, pages 235–52. Kluwer, 1994.

[96] Tim Sutton and David Webb. *FibreChannel: The Digital Highway Made Practical*, 22 October 1994.

[97] VERITAS Software Corporation. Product Comparison of VERITAS On-Line Storage Management 2.1 to SunSoft Solstice DiskSuite 4.0. http://www.veritas.com. VERITAS Software Corporation, 21 June 1996.

[98] VERITAS Software Corporation. File System White Paper, part I, II and III. http://www.veritas.com. VERITAS Software Corporation, November 1996.

[99] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-Time Mach: Towards a predictable Real-Time Systems. *Proceedings of USENIX Mach workshop*. USENIX, 1990.

[100] Sape J. Mullender, Ian M. Leslie, and Derek McAuley, *Operating System Support for Distributed Multimedia*, *Usenix 1994 Summer Conference* (Boston). Usenix, June 1994.

[101] Andrew Heybey, Mark Sullivan, and Paul England, *Calliope: A Distributed, Scalable, Multimedia Server*, *USENIX 1996 Annual Technical Conference* (San Diego, CA, January 1996). USENIX Association, January 1996.

[102] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, **34**(4):30–44, April 1991.

[103] Yuewei Wang, Jonathan C.L. Liu, David H.C. Du, and Jenwei Hsieh. Efficient video file allocation schemes for video-on-demand services. *Multimedia Systems*, **5**(5):283–96, September 1997.

[104] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.

Acknowledgements

I would like to thank everybody who has helped me write this thesis. In particular I would like to thank Sape Mullender for his support and criticism during the enumerable discussions in the train. Also, I would like to thank Richard Golding at HP Laboratories for his support and criticism.

Further I would like to thank my friends at work, coworkers and the colleagues from the Pegasus project for their influence on this work. In particular I would like to thank Feico Dillema because it is his influence to point me at the scheduling opportunities. Also, I would like to thank Frans Jonkers for measuring the Seagate Cheetah disks.

I would like to thank the Nederlandse Spoorwegen for their real-time behavior with only a few deadline misses and Deutsche Bahn for providing beer every Friday.

I would like to thank Annebelle for keeping me busy with everything but my thesis on Tuesdays. Her favorite line was ‘Eerst gaat papa op de computer spelen, dan ik.’

Publications

[1] Peter Bosch, *Inheritance vs. delegation: Is one better than the other?*, Universiteit Twente, April 1993.

[2] Peter Bosch, Sape Mullender and Tage Stabell-Kulø, *Huygens File Server and Storage Architecture*, BROADCAST - First Year Report, volume 3. ESPRIT BRA 6360, October 1993.

[3] Peter Bosch, *A Cache Odyssey*, M.Sc. thesis, SPA-94-10, Universiteit Twente, June 1994, Also published as Pegasus paper 94-07, ESPRIT BRA 6865.

[4] Peter Bosch, Sape Mullender, *Extensive Write Caching*, BROADCAST - Second Year Report, volume 3, ESPRIT BRA 6360, November 1994.

[5] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes, *Idleness is not sloth*, Conference proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems (New Orleans), pages 201–12, Usenix Association, 16–20 Januari 1995.

[6] Peter Bosch, and Sape J. Mullender, *PFS and Patsy: a new file system design methodology*, Broadcast 3rd year deliverable, July 1995.

[7] Peter Bosch, and Sape J. Mullender, *Cut-and-Paste file-systems: integrating simulators and file-systems*, Conference proceedings of the USENIX 1996 Technical Conference on UNIX and Advanced Computing Systems (San Diego), Usenix Association, Januari 1996.

[8] Peter Bosch, and Sape J. Mullender, *PFS: A Distributed and Customizable File System*, Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems (Seattle), IEEE, October 1996.

[9] Richard Golding, Peter Bosch, and John Wilkes, *Idleness is not sloth*, Hewlett-Packard TR–96–140, October 1996.

[10] Peter Bosch, and Sape J. Mullender, *Nihilistic: A continuous media file system for recording and playback of variable bit rate streams*, Pegasus-II 1st year deliverable 3.1.1, July 1997.

[11] Peter Bosch, and Sape J. Mullender, *Pegasus deliverable 3.2.1: Posix interfaces for Clockwise*, Pegasus-II 2nd year deliverable, Februari 1998.

[12] Peter Bosch, *Using PCs as continuous media servers*, Conference proceedings of the NLUUG Voorjaarsconferentie 1998, pages 115–129, Ede, Netherlands, 13 May 1998.

[13] Peter Bosch *Pegasus deliverable: NFS on VFS/Clockwise*, Pegasus-II 2nd year deliverable, June 1998.

[14] Peter Bosch, and Sape J. Mullender, *"Don't hide power"*, Conference proceedings of the ACM SIGOPS European Workshop 1998, Sintra, Portugal, September 1998.

[15] Peter Bosch *Pegasus deliverable 2.2.2: Profiling tools for Nemesis*, Pegasus-II 2nd year deliverable, October 1998.
[16] Peter Bosch, Sape J. Mullender, Pierre G. Jansen, *Clockwise: A Mixed-Media File System*, Conference proceedings of the IEEE ICMCS'99 (to appear), Firenze, Italy, June 1999.

Post script

I was born in Amsterdam on September 21st, 1966. Except for a short stay in Mountain View, California, I have lived all my life in Amsterdam.

Many people have asked me over the last 7 years why I have never moved to Twente, but instead have subjected myself to the idiosyncratic behavior of the Nederlandse Spoorwegen and Deutsche Bahn. The reason is actually quite simple: traveling long distances by train is relaxing and the best place to program software or write documents. Most of code for Clockwise and this entire thesis are written in 'hondenkoppen', 'koplopers' and the 'Havelsee.'

Knowing myself, I would never have had the patience to analyze performance results if I would not be forced to sit in a train for four hours a day. From this perspective, IP connectivity in the train, or even simple GSM technology are developments that do not work for me. Instead, I appreciate the time that I am 'incommunicado.'

The other reason for living in Amsterdam is because Amsterdam is a fun place to live. I do not believe that there are other cities in the Netherlands that are so diverse as Amsterdam: the wide range of restaurants, bars, (movie) theaters make a city worthwhile to live in. Surely, Amsterdam has its bad sides, but the good things outweigh the bad things. Please note that I am not saying that Twente is not a good place to live – I just think it is not the place for me.

As for the biography that is requested by the Ph.D. thesis protocol, here it goes. Education: HAVO, HTS in Amsterdam, VU Amsterdam combined with the UT in Enschede. Experience: OSL Integrators in Almere, CWI in Amsterdam, ACE in Amsterdam, HP Laboratories in Palo Alto, California, and the UT in Enschede.